The Pennsylvania State University
The Graduate School
College of Information Sciences and Technology

OPAQUE PREDICATE: ATTACK AND DEFENSE IN OBFUSCATED BINARY CODE

A Dissertation in
Information Sciences and Technology
by
Dongpeng Xu

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2018

The dissertation of Dongpeng Xu was reviewed and approved* by the following:

Dinghao Wu
Associate Professor of Information Sciences and Technology
Dissertation Advisor, Chair of Committee

Peng Liu
Professor of Information Sciences and Technology
Committee member

Sencun Zhu
Associate Professor of Computer Science and Technology
Committee member

Trent Jaeger
Professor of Computer Science and Technology
Outside member

Andrea Tapia
Associate Professor of Information Sciences and Technology
Director of Graduate Programs

*Signatures are on file in the Graduate School.

Abstract

An opaque predicate is a predicate whose value is known to the obfuscator but is difficult to deduce. It can be seamlessly applied together with other obfuscation methods such as junk code to turn reverse engineering attempts into arduous work. Opaque predicates have been widely used in various areas of software security such as software protection, software watermarking, obfuscation, and metamorphic malware. The arms race between the construction and detection of opaque predicates is an interesting topic in computer security area.

This thesis introduces new attack and defense techniques about opaque predicates in binary code. First, a logic oriented opaque predicate detection tool called LOOP is proposed. By conducting symbolic execution along a trace, LOOP constructs general logical formulas to represent the intrinsic characteristics of opaque predicates. The formulas are then solved by a constraint solver and the result answers whether the predicate under examination is opaque or not. Besides, LOOP is obfuscation resilient and able to detect previously unknown opaque predicates. Our experimental result demonstrates LOOP is effective and efficient. By integrating LOOP with code normalization for matching metamorphic malware variants, we show that LOOP is an appealing complement to existing malware defenses.

Second, a new control flow obfuscation scheme called generalized dynamic opaque predicate is proposed. We extend the conventional concept of dynamic opaque predicate to common program structures (e.g., straight-line code, branch, and loop). Besides, our new design does not require dynamic opaque predicates to be strictly adjacent, which is more resilient to deobfuscation techniques. The evaluation result shows generalized dynamic opaque predicates overcome the limitations in conventional opaque predicates.

Third, we propose a novel technique called bit-precise symbolic loop mapping to identify cryptographic functions in obfuscated binary code. Advanced opaque predicates adopt cryptographic functions to challenge existing opaque predicate detection methods. Our trace-based approach captures the semantics of possible cryptographic algorithms with bit-precise symbolic execution in a loop. Then we

perform guided fuzzing to efficiently match boolean formulas with known reference implementations. We have developed a prototype called CryptoHunt and evaluated it with a set of obfuscated synthetic examples, well-known cryptographic libraries, and malware. Compared with the existing tools, CryptoHunt is a general approach to detecting commonly used cryptographic functions such as TEA, AES, RC4, MD5, and RSA under different control and data obfuscation scheme combinations.

Our research deepens and expands the knowledge of attack and defense techniques about opaque predicates. We are the first to give a systematic categorization of opaque predicates and a method for the detection of dynamic opaque predicates. Our research about the extension of dynamic opaque predicates greatly widens its application. The cryptographic algorithm detection can be applied to various fields, such as ransomware detection, malware analysis, and software reverse engineering.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Firstly, I would like to express my special appreciation and thanks to my advisor Prof. Dinghao Wu for the continuous support of my Ph.D. study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of my research, communication, and career development. I appreciate all his contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating. I could not have imagined having a better advisor and mentor for my Ph.D. study.

I would also like to thank the rest of my thesis committee: Prof. Peng Liu, Prof. Sencun Zhu, and Prof. Trent Jaeger, for their insightful comments and suggestions, but also for the questions which encourage me to improve and widen my research from different perspectives.

I thank my fellow labmates for the stimulating discussions, for every project we were working on day and night, and for all the fun and difficulties we have had in the last five years. In particular, I am grateful to my friend and colleague, Dr. Jiang Ming, for providing invaluable help on both my research and life.

Last but not the least, I would like to thank my family, for always supporting and encouraging me spiritually in my life. The last word of acknowledgment I have saved for my dear wife Shan Jing, who is always my support in the moments when there was no one to answer my queries.

This research was supported in part by the National Science Foundation (NSF) under grants CNS-1223710, CCF-1320605, and CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-13-1-0175, N00014-16-1-2912, N00014-16-1-2265, and N00014-17-1-2894.

Chapter 1 Introduction

Nowadays as large numbers of computers and smart phones are connected to the Internet, software has become a target of attackers, copyright pirates, and malware developers. As a result, software users and developers pay more and more attention on software protection. One of the most important techniques in software protection is software obfuscation. Generally speaking, software obfuscation is the act of creating source or machine code that is difficult for human to understand. After the concept was first introduced by the research from Collberg et al. [1–3], varieties of software obfuscation methods have been proposed [4–8]. While the technique is originally proposed to protect benign programs, it has been broadly used by malware developers to help malware evade from detection [9–12]. Detection of these obfuscated malware becomes a challenging problem in modern malware analysis.

1.1 Opaque Predicate

Among software obfuscation techniques, opaque predicate is a simple and effective method. In computer science, predicates are conditional expressions that evaluate to true or false. An opaque predicate means its value are known to the obfuscator at obfuscation time, but it is difficult for an attacker to figure it out afterwards. Used together with junk code, the effect of opaque predicates results in a heavily cluttered control flow graph with redundant infeasible paths. Therefore, any further analysis based on the control flow graph will turn into arduous work. Compared with other control flow graph obfuscation methods such as control flow flattening [13] and call stack tampering [14], opaque predicates are more stealthy because it is difficult to

differentiate opaque predicates from original path conditions in binary code [15, 16]. Also, another benefit of opaque predicates is they have little impact on the runtime performance and code size. First proposed by Collberg et al. [1], opaque predicates have been applied widely in various ways, such as software diversification [17, 18], metamorphic malware mutation [19,20], software watermarking [21,22], and Android Apps obfuscation [23]. Due to the low-cost and stealthy properties, most real-world obfuscation toolkits have supported inserting opaque predicates into a program, through link-time program rewriting or binary rewriting [24–26].

Real-world obfuscation tools have already supported embedding opaque predicates into programs at link time or binary level [24–26]. As a result, control flow graph is heavily cluttered with infeasible paths and software complexity increases as well [27]. Unlike other control flow graph obfuscation schemes such as call stack tampering or control flow flattening [13], opaque predicates are more covert as it is hard to distinguish opaque predicates from normal conditions. Furthermore, opaque predicates can be seamlessly woven together with other obfuscation methods, such as opaque constants [5] and metamorphic mutations [28] to further subvert reverse engineering efforts. One example can be found in a recent notorious "0 day" exploit (CVE-2012-4681), in which opaque predicates are used together with encrypted code [29]. Therefore, it has become more difficult to locate the exploit code of interest due to the use of opaque predicates.

1.2 Arms Race

In computer security, arms race happens commonly between two counter techniques, such as opaque predicate construction and detection in this thesis. In terms of attack and defense, we use the term "attack" to refer to the work to construct more effective opaque predicates, because opaque predicates are widely adopted by malware developers to hide their malware. In contrast, we use "defense" to refer to those work to detect opaque predicates inside software. Those detection methods help security analysts defend against obfuscated malware.

1.2.1 Attack

From the attacker's perspective, researchers propose varieties of methods for constructing opaque predicates with low runtime cost and resilience to deobfuscation. Existing opaque predicates can be classified into three categories.

The first category is called *invariant opaque predicates*. These predicates are always evaluated to the same value for all possible inputs. Invariant predicates are mainly constructed from well-known algebraic theorems [21, 22]. For example, predicate ($x^3 - x \equiv 0 \pmod 3$) is opaquely true for all integers x. Since it is easy to construct invariant opaque predicates they are commonly used. The second category, *contextual opaque predicates*, is built on some program invariant under a specific context. That means only the obfuscator knows such a predicate is true (false) at a particular point, but could be false (true) if the context is not satisfied. The third category, *dynamic opaque predicates*, is the most advanced one. In this category, a set of *correlated* and *adjacent* predicates evaluate to the same value in any given run, but the value might be different in another run. In any case, the program produces the same output. To make matters worse, dynamic opaque predicates can be carefully crafted by utilizing the static intractability property of pointer aliasing [1]. In addition, cryptography has been adopted to enhance opaque predicate [30].

1.2.2 Defense

On the other hand, as the defense technique, opaque predicate detection has attracted many security researchers' attention. A number of methods have been proposed to identify opaque predicates [1, 31–34]. Collberg et al. [1] propose a heuristics based method, which is able to detect a specific type of already known opaque predicates. Preda et al. [32] use abstract interpretation to defend specific opaque predicates, such as $\forall x \in \mathbb{Z} : n | f(x)$. Madou [31] and Udupa et al. [34] did research on opaque predicate detection based on the fact that the value of opaque predicate doesn't change during multiple executions. The invariant property of those "static" opaque predicates leads to the fact that they are likely to be detected by program analysis tools. However, all existing detection approaches only focus on invariant opaque predicates. There has been little work on systematically modeling and solving contextual or dynamic opaque predicates. Detection of

opaque predicates that are enhanced by cryptography is also a challenge for security analysts.

1.3 Thesis Organization

In this thesis, we mainly present three research work on the attack and defense techniques about opaque predicate obfuscation. The first one is a defense technique, which detects opaque predicate based on logic formulas. The second one is an attack method called generalized dynamic opaque predicate. The third work is a defense technique, which can detect cryptographic functions inside obfuscated binary code.

The rest of the thesis is organized as follows. Chapter 2 shows a high level picture of the three methods. Chapter 3 provides the background knowledge and related work about opaque predicate and cryptographic function detection. Chapter 4 presents the detailed design and evaluation of the logic based opaque predicate detection. Chapter 5 shows the design of generalized opaque predicate in depth and the experiment result. Chapter 6 presents the defense work to detect cryptographic function inside obfuscated binary code. We discuss the limitations and future work in Chapter 7 and conclude the thesis in Chapter 8.

Chapter 2
Overview

This chapter provides a quick tour of the three research work in this thesis. In terms of attack and defense, the first and the third work are defense methods, which help detect opaque predicates. The second work is an attack method, which constructs advanced opaque predicates.

2.1 Logic-Oriented Opaque Predicate Detection

The first research work, *LOOP*, is a new logic-based, general method to detect opaque predicate in binary code. This work fills in the blank of detecting contextual and dynamic opaque predicate.

In general, we first perform symbolic execution on an execution trace to build path condition formulas, on which we detect invariant opaque predicates (the first category) by verifying tautologies with a constraint solver. In the next step, we identify an implication relationship to detect possible contextual opaque predicates (the second category). Finally, with input generation and semantics-based binary diffing techniques, we further identify correlated predicates to detect dynamic opaque predicates. Our method is based on formal logic that captures the intrinsic semantics of opaque predicates. Hence, LOOP can detect previously unknown opaque predicates. A benefit of LOOP's trace oriented detection is that it is resilient to most of the known attacks that impede static analysis, ranging from indirect jump, pointer alias analysis [35], opaque constants [5], to function obfuscation [36]. Our results can be fed back to security analysts to further de-obfuscate the cluttered control flow graph incurred by opaque predicates.

We have implemented LOOP to automate opaque predicate detection on top of the BAP platform [37] and conducted the evaluation with a set of common utilities and obfuscated malicious programs. The experimental results show that LOOP is effective and general in detecting opaque predicates with zero false negatives. Several optimizations such as taint propagation and "short cut" strategy offer enhanced performance gains. To confirm the merit of our approach, we also test LOOP in the task of code normalization for metamorphic malware [19,20]. This kind of malware often uses opaque predicates to mutate the code during propagations to evade signature-based malware detection. The result indicates that LOOP can greatly speed up control flow graph matching by a factor of up to 2.0.

2.2 Generalized Dynamic Opaque Predicate

In the second research work, we present a systematic design of a new control flow obfuscation method, *Generalized Dynamic Opaque Predicates*, which is able to inject diversified dynamic opaque predicates into complicated program structures such as branches and loops. Being compared with the previous technique which can only insert dynamic opaque predicates into straight-line program, our new method is more resilient to program analysis tools. We have implemented a prototype tool based on the LLVM compiler infrastructure [38]. The tool first performs fine-grained data flow analysis to search possible insertion locations. After that it automatically transforms common program structures to construct dynamic opaque predicates. We have tested and evaluated the tool by obfuscating several hot functions of GNU core utilities with different obfuscation levels. The experimental results show that our method is effective and general in control flow obfuscation. Besides, we demonstrate that our obfuscation can defeat the commercial binary difference analysis tools and the state-of-the-art formal program semantics-based deobfuscation methods. The performance data indicate that our proposed obfuscation only introduces negligible overhead.

2.3 Crypto Function Detection in Obfuscated Binary Code

In order to defend against the opaque predicate enhanced by cryptography, we propose a new method, *CryptoHunt*, to detect cryptographic functions inside obfuscated binary code. Our key idea is to capture the fine-grained semantics of the principal cryptographic transformation iterations along an execution trace. The execution trace is further split into segments according to an enhanced loop abstraction. We then perform bit-precise symbolic execution inside a loop body, and the generated boolean formulas are later used as signatures to efficiently match cryptographic algorithms in obfuscated binaries. Our core technique, *bit-precise symbolic loop mapping*, is effective to revert various data and control obfuscation effects, and also with a much broader detection scope.

We have evaluated CryptoHunt on a set of synthetic examples collected from GitHub, well-known cryptographic libraries, and malware. We compared CryptoHunt with other six representative tools, and the experiment results are encouraging. In all cases, only CryptoHunt is able to detect commonly used cryptographic functions (e.g., TEA, AES, RC4, MD5, and RSA) under different control and data obfuscation scheme combinations.

2.4 Contributions

In summary, we make the following contributions in this thesis.

1. We study the common limitations of existing work in detecting opaque predicates and propose LOOP, an effective and general approach that identifies opaque predicates in obfuscated binary code. LOOP is based on strong principles of program semantics and logic, and can detect known and unknown, simple invariant, intermediate contextual, and advanced dynamic opaque predicates.

2. LOOP is developed based on symbolic execution and theorem proving techniques. Our evaluation shows that LOOP automatically diagnoses opaque predicates in an execution trace with zero false negatives.

3. LOOP is the first solution towards solving both contextual and dynamic opaque predicates.

4. We propose an effective and generalized opaque predicate obfuscation method. Our method outperforms existing work by automatically inserting opaque predicates into more general program structures like branches and loops, whereas previous work can only work on straight-line code.

5. We demonstrate generalized dynamic opaque predicate is very resilient to the state-of-art opaque predicate detection tool.

6. We propose a new method called CryptoHunt to detect cryptographic functions in obfuscated binaries. Our key solution is to match the principal cryptographic transformation iterations with bit-precise symbolic loop mapping. CryptoHunt exhibits stronger resilience to code obfuscation techniques and a wider detection range than previous methods.

7. We design a guided fuzzing method to solve the scalability issue of bit-wise symbolic formula equivalence checking. Our approach greatly reduces the number of possible matches, and can be applied to speed up other semantics-based binary difference analysis methods.

8. We have implemented prototype systems for each of the three work and the source code is publicly available.

Chapter 3
Background and Literature Review

3.1 Code Obfuscation

Code obfuscation techniques, which are first designed to protect software intellectual property [1], deliberately transform code to make it more difficult to understand. Nowadays malware authors also heavily rely on code obfuscation to evade detection [39]. For example, one frequently used obfuscation technique in malware is binary packing [14], which first compresses or encrypts an executable binary into data and then recover the original code when the packed program starts running. In general, obfuscation techniques can be classified as two pervasive methods: control obfuscation and data obfuscation. Control obfuscation, such as control flow flattening [40] and opaque predicate [2], greatly changes control flow information to impede reverse engineering. Data obfuscation is intended to conceal data value and usage. For example, data encoding schemes [41,42] convert a variable representation to an obscure one, while data aggregation [43] changes how a variable or array is aggregated. Recovering high-level data abstractions and types from binary code is already pretty hard [44, 45], and data obfuscation will make it more challenging.

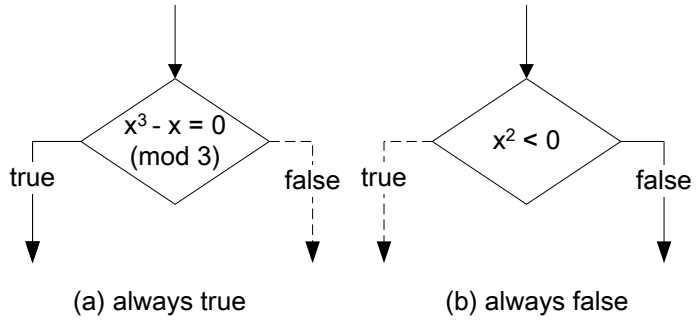


Figure 3.1: Examples of two invariant opaque predicates for all integers x.

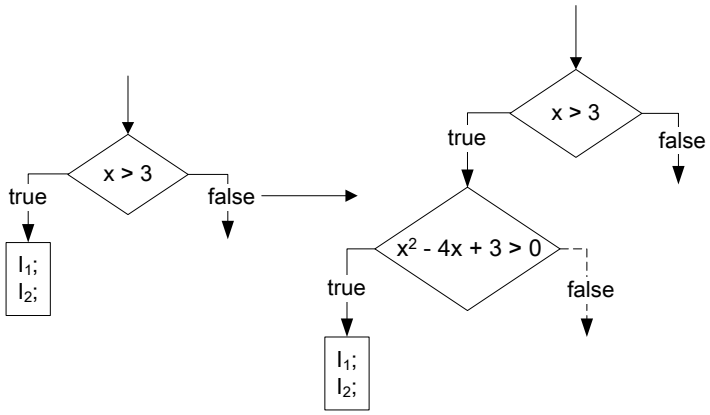


Figure 3.2: Example of a contextual opaque predicate for all integers satisfying $x > 3$.

3.2 Opaque Predicate Categories

3.2.1 Invariant Opaque Predicates

An opaque predicate is invariant when its value always evaluates to true or false for all possible inputs, but only the obfuscator knows the value in advance. Figure 3.1 shows two cases of invariant opaque predicates: always true and always false. The dashed line indicates that the path will never be executed. Due to the simplicity, this kind of opaque predicates have a large set of candidates. Most of them are derived from well-known algebraic theorems [22] or quadratic residues [21]. However, the invariant property also becomes the drawback of this category. For example, we can identify possible invariant opaque predicates by observing the branches that never change at run time with fuzzing testing [31].

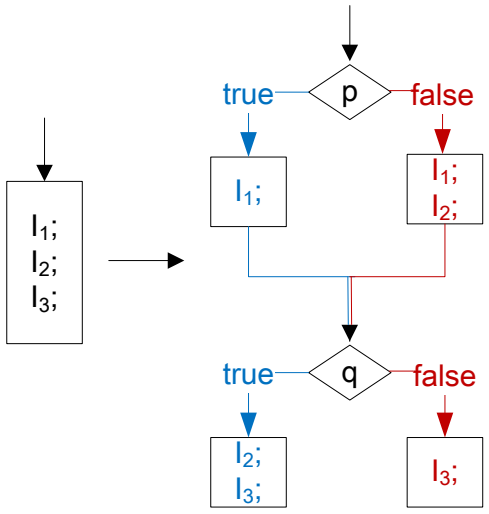


Figure 3.3: Example of a dynamic opaque predicate.

3.2.2 Contextual Opaque Predicates

To avoid an opaque predicate always produces the same value for all inputs, Drape [46] proposes a more covert opaque predicate that is always true (false) under a specific precondition, but could be false (true) when precondition does not hold. We call this kind as contextual opaque predicates, which can be carefully constructed based on program invariants under a particular context. Figure 3.2 shows an example of contextual opaque predicate, in which $x^2 - 4x + 3 > 0$ is always true if the precondition $x > 3$ holds. Note that the constant value in the precondition can be further obfuscated [5] to hide the context relationship.

3.2.3 Dynamic Opaque Predicates

Palsberg et al. [47] introduce the idea of dynamic opaque predicates, which are a family of *correlated* and *adjacent* predicates that all present the same value in any given run, but the value may vary in another run. That means the values of such opaque predicates switch dynamically. Combined with code clone, dynamic opaque predicates can always produce the same output. The term "correlated" is used to describe that dynamic opaque predicates contain a set of mutually related predicates, and "adjacent" means these opaque predicates execute one after another strictly. Figure 3.3 illustrates an example of dynamic opaque predicates. Two correlated predicates, p and q, meet the requirement of evaluating to true (false)

in any given run. The original three instructions $\{I_1; I_2; I_3;\}$ execute one after another. After transformation, each run either follows the path $p \wedge q$ (blue path) or $\neg p \wedge \neg q$ (red path). In any case, the same instructions will be executed. Look carefully at Figure 3.3, we can find another common feature. Since predicate q divides both blue path and red path into different segments (i.e., $\{I_1;\}$ vs. $\{I_1; I_2;\}$ and $\{I_2; I_3;\}$ vs. $\{I_3\}$), p and q must be strictly adjacent; or else the transformation is not semantics-persevering. The correlated predicates can be crafted by utilizing pointer aliasing, which is well known for its static intractability property [1].

Existing efforts in identifying opaque predicates mainly focus on invariant opaque predicates and they are unable to detect more covert opaque predicates such as contextual and dynamic opaque predicates. A general and accurate approach to opaque predicate detection is still missing. Our research aims to fill in this gap.

3.3 Opaque Predicate Enhanced by Cryptography

Zobernig et al. [30] adopt cryptographic functions to enhance existing opaque predicate. They borrow the idea from using cryptographic hash function to protect password check program. Similarly, they use hash functions to hide the opaque predicate, which is a constant comparison function as follows. Equation 3.1 shows a normal opaque predicate, in which $P(x)$ always equals to the constant C. The opaque predicate can be enhanced by a cryptographic hash function $H()$ as shown in Equation 3.2. The same hash function $H()$ is applied to $P(x)$ and C, so the result must still be equivalent. The computational complexity of the hash function $H()$ guarantees that no automated theorem prover is able to solve the equation.

$$P(x) = C \tag{3.1}$$

$$H(P(x)) = H(C) \tag{3.2}$$

Cryptographic opaque predicate challenges all opaque predicate detection methods based on theorem prover and auto testing techniques. In this thesis, we propose a new technique to detect cryptographic functions inside obfuscated binary code. This method helps identify cryptographic opaque predicate inside binary code.

3.4 Opaque Predicate Detection

The concept of opaque predicates is first proposed by Collberg et al. [1] to prevent malicious reverse engineering attempts. Collberg et al. [1] also provide some ad-hoc detection methods. One of them is called "statistical analysis"; that is, a predicate that always produces the same result over a larger number of test cases has a great chance to be an opaque predicate. Due to the limited set of inputs, statistical analysis could lead to high false positive rates. Preda et al. [32] propose to detect opaque predicates by abstract interpretation. However, their approach only detects a specific type of known invariant opaque predicates such as $\forall x \in Z.n|f(x)$. Madou [31] first identifies candidate branches that never change at run time, and then verifies such predicates by fuzz testing with a considerably high error rate. Udupa et al. [34] utilize static path feasibility analysis to determine whether an execution path is feasible. However, their approach cannot work on a highly obfuscated binary with, for example, complicated opaque predicates based on pointer aliasing, which is known to be statically intractable. OptiCode [33] has a similar idea in using theorem prover to decide if a desired branch is always true or false, but it can only deal with invariant opaque predicates. Our work is different from the previous work in that LOOP is both general and accurate. We are able to detect previously unknown opaque predicates in obfuscated binary, including more sophisticated ones such as contextual and dynamic opaque predicates.

3.5 Infeasible Path Identification

The effect of opaque predicates is to obfuscate control flow graph with infeasible paths. In software testing, eliminating infeasible paths saves efforts to generate redundant test cases. Previous work identifies infeasible paths in source code, either by branch correlation analysis [48], pattern matching [49], or monitoring the search for test data [50]. However, these work cannot be directly used to detect opaque predicates in an adversary environment, in which the program source code under examination is typically absent. Therefore, we believe LOOP has compelling application in identifying infeasible paths in binary.

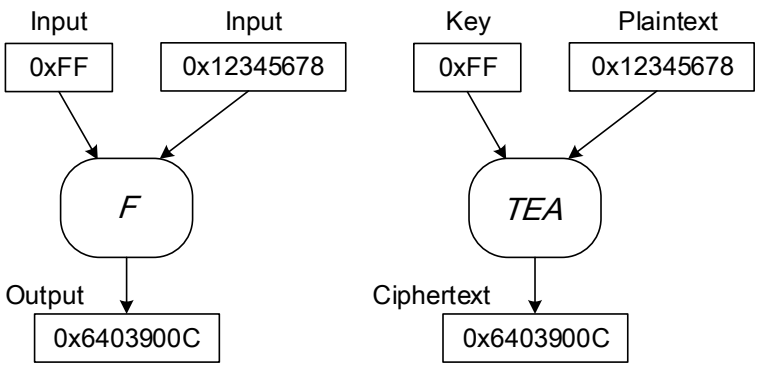


Figure 3.4: Since F has the same input-output mapping with TEA algorithm, we can recognize F as TEA with an overwhelming probability.

3.6 Cryptographic Function Detection

3.6.1 Static Method

Static crypto detection methods detect cryptographic functions in binaries prior to execution. They perform static analysis to recognize code/data features. Lutz's work [51] recognizes cryptographic code via three heuristics, such as the presence of loops, entropy, and a high ratio of bitwise operations. Wang et al. [52] utilize a similar method to identify the message decryption phase so as to locate the encrypted data. Matenaar et al. [53] apply multiple detection heuristics such as entropy, constant value, and crypto API. Lestringant et al. [54] utilize data flow graph as the signature to identify symmetric cryptographic algorithms. Static detection has no runtime overhead and is sufficient for unobfuscated programs. However, static visible signatures can be easily camouflaged by code obfuscation techniques [5]. Calvet et al. [55] have demonstrated a very lightweight data obfuscation scheme (splitting a const value into two smaller numbers) can fail static detection.

3.6.2 Dynamic Method

Dynamic detection searches visible cryptographic algorithm features at run time. Compared with the pre-execution tools, dynamic approaches are more accurate since it follows the real execution path and knows the actual dynamic state.

Therefore, dynamic detection is widely applied to analyze obfuscated malware. CipherXRay [56] detects cryptographic operations by observing data avalanche effect, which refers to a property of cryptographic algorithms such that a slight change in the input would cause significant changes in the output. However, CipherXRay is still based on some intuitive observations, which cannot detect the exact cryptographic algorithm used. Furthermore, stream ciphers neither show such avalanche effect. Gröbert et al. [57] first propose a reliable dynamic approach by mapping cryptographic function input-output (I/O) relations. They first aggregate contiguous memory accesses to form input and output parameters and then find whether there is an *exactly the same I/O mapping* with a known cryptographic function (see Figure 3.4). Aligot [55] extends this idea by automatically identifying and extracting parameters at a loop boundary. It also performs an inter-loop data flow analysis so as to better catch the parameter candidates. Then, Aligot also checks whether there exist a *perfect match* between loop I/O mapping and a reference implementation.

Since all the methods that rely on identifying unique input-output relations [55, 57, 58] treat a series of cryptographic operations as a "black box", they can tolerate code obfuscation and different implementations that happen within the "black box". Their detection effects ultimately depend on three key assumptions: 1) accurately locate the boundary where they want to compare I/O mappings with golden implementations (e.g., identify the scope of F in Figure 3.4); 2) precisely recover I/O parameters from memory (e.g., extract the input and output values); 3) F in Figure 3.4 must have a perfect match. However, a skilled attacker can easily break down these assumptions. For example, the smallest parameter size that current approaches can extract is one byte. Any data obfuscation scheme that aggregates a non one-byte multiples variable (e.g., a 15-bit length variable in Figure 6.5) can complicate parameter extraction. Also, Base64 encoding is commonly found in malware to disguise their malicious payloads [42], which can convert I/O parameter values to different ones and fail the I/O mapping eventually. And even worse, malware authors have already customized non-standard cryptographic algorithm implementations [59, 60] so that F in Figure 3.4 produces a different output. In contrast, our approach inherits dynamic analysis advantages and take F as a "gray box" by representing I/O mappings with bit-precise symbolic execution, which is effective to beat both code obfuscation and non-standard implementations.

3.7 Symbolic Execution

Being first proposed by King [61], symbolic execution is an effective technique in the program analysis field. Briefly speaking, symbolic execution replaces concrete values in a program with symbolic values and simulates the execution of the program so that all variables hold symbolic expressions. Symbolic execution has emerged as a fundamental approach for reasoning software security problems [62–64]. EXE [65] automatically detects bugs in C code. KLEE [66] is capable of automatically generating test cases that achieve high path coverage. BAP platform [37], the successor of BitBlaze [67], provides binary code symbolic execution and verification functions. We also perform symbolic execution to model the semantics of a loop body. However, our approach reveals a distinct design choice: CryptoHunt's symbolic execution contains only one atomic data type, *boolean*. CryptoHunt substitutes each loop input variable as a set of bit-symbols and represents loop input-output relations as multiple boolean formulas. Suppose we want to find whether two 32-bit symbolic variables are equivalent, instead of matching two whole 32-bit vectors, we compare them bit-by-bit. In this way, we can find the fact that, for example, the low 15-bit of these two variables are matched. Our solution ensures that we can accurately capture data obfuscation effects.

3.8 Concolic Testing

Our logic-based approach is inspired by the active research in concolic testing [62, 65, 66, 68], a hybrid software verification method combining concrete execution with symbolic execution. Similar to SAGE [62], LOOP first maps symbols to inputs and then collects constraints of these symbolic inputs along a recorded execution trace. The difference is our primary purpose is not for path exploration; instead we construct formulas representing the characteristics of opaque predicates and solve these formulas with a constraint solver. In addition to automatic input generation, we have seen applications of concolic testing in discovery of deviations in binary [69], software debugging with golden implementation [70], and alleviating under-tainting problem [71]. Our approach adopts part of the concolic testing idea in software deobfuscation and malware analysis.

3.9 Binary Difference Analysis

Another related field to our work is automatically finding semantic differences/similarities in binaries [72–79], which has a wide application in practice, such as malware lineage inference [72,78], software plagiarism detection [74,79] and cross-architecture bug search [76, 77]. CryptoHunt differs from this previous work in a number of ways. First, CryptoHunt is specifically designed to detect cryptographic function reusing in obfuscated binaries, and a cryptographic function typically occupies a small fraction of binary code. Most of the previous work more or less relies on static features, such as control flow graph [73, 74, 76, 77, 79] and identifying function in stripped binaries [75], which make them not competent to our task. Second, much previous work also compares binaries with symbolic execution and constraint solving [72–74, 78, 79]. But they suffer from high performance penalty due to excessive symbolic variable mapping. To relieve this performance bottleneck, we propose a guided fuzzing approach to filter out large numbers of impossible matches.

Chapter 4
Logic-Oriented Opaque Predicate Detection in Binary Code

4.1 Introduction

In this chapter, we introduce a new logic-based, general approach to detecting opaque predicates progressively in obfuscated binary code. We first perform symbolic execution on an execution trace to build path condition formulas, on which we detect invariant opaque predicates (the first category) by verifying tautologies with a constraint solver. In the next step, we identify an implication relationship to detect possible contextual opaque predicates (the second category). Finally, with input generation and semantics-based binary diffing techniques, we further identify correlated predicates to detect dynamic opaque predicates. Our method is based on formal logic that captures the intrinsic semantics of opaque predicates. Hence, LOOP can detect previously unknown opaque predicates. A benefit of LOOP's trace oriented detection is that it is resilient to most of the known attacks that impede static analysis, ranging from indirect jump, pointer alias analysis [35], opaque constants [5], to function obfuscation [36]. Our results can be fed back to security analysts to further de-obfuscate the cluttered control flow graph incurred by opaque predicates.

We have implemented LOOP to automate opaque predicates detection on top of the BAP platform [37] and conducted the evaluation with a set of common utilities and obfuscated malicious programs. The experimental results show that LOOP

The work of this chapter is published in the 22nd ACM Conference on Computer and Communications Security [80].

is effective and general in detecting opaque predicates with zero false negatives. Several optimizations such as taint propagation and "short cut" strategy offer enhanced performance gains. To confirm the merit of our approach, we also test LOOP in the task of code normalization for metamorphic malware [19, 20]. This kind of malware often uses opaque predicates to mutate the code during propagation to evade signature-based malware detection. The result indicates that LOOP can greatly speed up control flow graph matching by a factor of up to 2.0.

In summary, we make the following contributions.

- We study the common limitations of existing work in detecting opaque predicates and propose LOOP, an effective and general approach that identifies opaque predicates in the obfuscated binary code. Our approach captures the intrinsic semantics of opaque predicates with formal logic, so that LOOP can detect previously unknown opaque predicates.
- Our method is based on strong principles of program semantics and logic, and can detect known and unknown, simple invariant, intermediate contextual, and advanced dynamic opaque predicates.
- LOOP is developed based on symbolic execution and theorem proving techniques. Our evaluation shows that our approach automatically diagnoses opaque predicates in an execution trace with zero false negatives.
- To the best of our knowledge, our approach is the first solution towards solving both contextual and dynamic opaque predicates.

The rest of this chapter is organized as follows. Section 4.2 illustrates our core method with a motivating example. Section 4.3 describes each step of our approach in detail. Section 4.4 introduces our implementation. We evaluate our approach in Section 4.5. Discussions and future work are presented in Section 4.6.

4.2 Overview

4.2.1 Method

The core of our approach is an opaque predicate detector, whose overall detection flow is shown in Figure 4.1. There are three rounds in our system to detect three

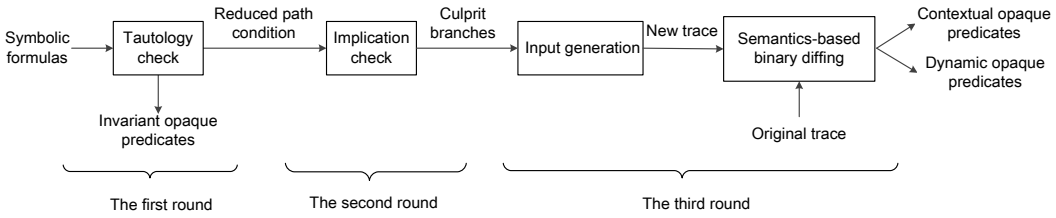


Figure 4.1: Opaque predicate detector.

kinds of opaque predicates progressively. Here we present an overview of our core method.

Since embedding opaque predicates into a program is a semantics-preserving transformation, deterministic programs before and after opaque predicate obfuscation should produce the same output. Let us assume the program P is obfuscated by opaque predicates and the resulting program is denoted as P_o. The logic of an execution of P_o is expressed as a formula Ψ, which is the conjunction of all branch conditions executed, including the following opaque predicates.

$$\Psi = \psi_1 \wedge ... \wedge \psi_{i-1} \wedge \psi_i \wedge ... \wedge \psi_n$$

Formula Ψ represents the conditions that an input must satisfy to execute the same path. Supposing constraint ψ_i is derived from an opaque predicate, we call ψ_i a *culprit branch*. The key to our approach is to locate all culprit branches in Ψ. Similar to dynamic symbolic execution [62] on binary code, we first characterize the logic of an execution in terms of symbolic path conditions, by performing a symbolic execution on the concrete execution trace.

Then our approach carries out three rounds of scanning. In the first round, we diagnose whether ψ_i is derived from an invariant opaque predicate by proving whether ψ_i is always true; that is, it is a *tautology*. Note that the false branch conditions have already been negated in the recorded trace. After that, we remove identified culprit branches from Ψ and continue to detect possible contextual opaque predicates in the second round. Our key insight is that a contextual opaque predicate does not enforce any *further* constraint on its prior path condition. Based on this observation, diagnosing whether a path constraint ψ_i $(1 \leq i \leq n)$ is a

```
 1 int opaque(int x)
 2 {
 3    int *p = &x;
 4    int *q = &x;
 5    int y = 0;
 6    if (x*x < 0)        // invariant opaque predicate
 7       x = x+1;
 8    if (x > 3)
 9    {                   // contextual opaque predicate
10       if (x*x-4x+3 > 0)
11       x = x<<1;
12     }
13    if ((*p)%2 == 0) // dynamic opaque predicate
14       y = x+1;
15    else
16    {
17       y = x+1;
18       y = y+2;
19    }
20    if ((*q)%2 == 0)
21    {
22       y = y+2;
23       x = y+3;
24    }
25    else
26       x = y+3;
27    return x;
28 }
```

Figure 4.2: A motivating example.

contextual opaque predicate boils down to answering an implication query, namely

$$\psi_1 \wedge ... \wedge \psi_{i-1} \Rightarrow \psi_i$$

Note that dynamic opaque predicates satisfy such implication check as well. For example, the combination of path condition for Figure 3.3 is either $p \wedge q$ or $\neg p \wedge \neg q$. It is straightforward to infer the following implication relationship.

$$(p \Rightarrow q) \wedge (\neg p \Rightarrow \neg q)$$

Assume we have detected $p \Rightarrow q$ in the second round of scanning. To further clarify p and q are correlated dynamic opaque predicates, we go one step further in the

```
 3  int *p = &x;
 4  int *q = &x;
 5  int y = 0;
 6  if (x*x < 0)        // invariant opaque predicate
 8  if (x > 3)
 9  {                   // contextual opaque predicate
10     if (x*x-4x+3 > 0)
11        x = x<<1;
12  }
13  if ((*p)%2 == 0) // dynamic opaque predicate
14     y = x+1;
20  if ((*q)%2 == 0)
21  {
22     y = y+2;
23     x = y+3;
24  }
27  return x;
28 }
```

Figure 4.3: An execution trace given x=4.

third round to verify whether $\neg p \Rightarrow \neg q$ holds as well. To this end, we automatically generate a new input that follows the path of $\neg p \wedge \neg q$. If $\neg p \Rightarrow \neg q$ is also true, we continue to compare trace segments guided by both $p \wedge q$ and $\neg p \wedge \neg q$ to make sure two paths are semantically equivalent. Further details about the detection process are discussed in Section 4.

4.2.2 Example

We create a motivating example (shown in Figure 4.2) to illustrate our core method. Figure 4.2 contains three different kinds of opaque predicates. Note that the two dynamic opaque predicates are constructed using pointer deference (line 13 $\sim$ line 27). The predicates in line 13 and line 20 are *correlated*, since they evaluate to the same value at any given run. In any case, the same instruction sequence $\{y = x + 1;\ y = y + 2;\ x = y + 3;\}$ will be executed. Consider an execution of the code snippet given $x = 4$ as input. Figure 4.3 shows a source-level view of such execution trace. We perform backward slicing and symbolic execution to calculate path condition formula Ψ. In our example, the predicates are represented as follows.

$$\psi_1 :\ x * x >= 0$$

$$\psi_2 : x > 3$$
$$\psi_3 : x * x - 4x + 3 > 0$$
$$\psi_4 : (*p)\%2 == 0$$
$$\psi_5 : (*q)\%2 == 0$$
$$\Psi : \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4 \wedge \psi_5$$

We present the three rounds step by step.

1. At the first round, we verify whether a predicate satisfies invariant property; i.e., it is a *tautology*. In our example, we prove that ψ_1 $(x * x >= 0)$ is always true and therefore conclude that ψ_1 is an invariant opaque predicate. After that, we remove ψ_1 from path condition Ψ to reduce the formula size and pass the new path condition to the next round.

2. We start the second round to detect possible contextual opaque predicates by performing implication check cumulatively from the first predicate. We identify two cases that satisfy the implication check in our example: $\psi_2 \Rightarrow \psi_3$ (ψ_1 has been removed), i.e., $(x > 3) \Rightarrow (x*x-4x+3 > 0)$ and $\psi_2 \wedge \psi_3 \wedge \psi_4 \Rightarrow \psi_5$, i.e., $(x > 3) \wedge (x * x - 4x + 3 > 0) \wedge ((*p)\%2 == 0) \Rightarrow ((*q)\%2 == 0)$. Note that ψ_5 in the second case is corresponding to the second culprit branch of the dynamic opaque predicates.

3. In the third round, we trace back from the culprit branches identified in the second step and further verify whether their prior predicates are correlated or not. Recall that another property of dynamic opaque predicates is being *adjacent*. In our example, we first negate each prior predicate as $\neg\psi_2$ and $\psi_2 \wedge \psi_3 \wedge \neg\psi_4$ and automatically generate inputs to satisfy such new path conditions. Here we generate two new inputs respectively, namely, $x = 0$ and $x = 5$. With the new traces, we perform implication check for $\neg\psi_2 \Rightarrow \neg\psi_3$ and $\psi_2 \wedge \psi_3 \wedge \neg\psi_4 \Rightarrow \neg\psi_5$. It is evident that $\neg\psi_2 \Rightarrow \neg\psi_3$ fails under the counterexample of $x = 0$. At last, we compare trace segments controlled by $\psi_4 \wedge \psi_5$ and $\neg\psi_4 \wedge \neg\psi_5$ to make sure they are semantically equivalent. As a result, we conclude that ψ_3 is a contextual opaque predicate and ψ_4 and ψ_5 consist of dynamic opaque predicates.

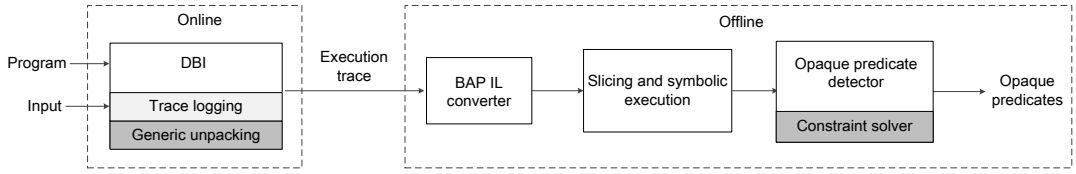


Figure 4.4: The architecture of LOOP.

For the presentation purpose, all the examples in this section are shown as C code and the predicates are presented as the "if" conditional statements. LOOP works at the binary level, in which the conditional statements such as "if" and "switch" are compiled as conditional jump instructions like je/jne/jg. LOOP identifies the conditional jump instructions that are opaquely true or false.

4.3 Approach

In this section, we present each step of our approach in detail. Figure 4.4 illustrates the architecture of LOOP, which includes two main parts: online trace logging and offline analysis. The online part, as shown in the left side of Figure 4.4, is built on top of a dynamic binary instrumentation (DBI) platform, enabling LOOP to work with unmodified binary code. To analyze packed malware, our online part includes two tools: generic unpacking and trace logging.

The logged trace is passed to the second part of LOOP for offline analysis (the right component of Figure 4.4). We first lift x86 instructions to BAP IL [37], a RISC-like intermediate language. Then starting from each predicate (branch), we perform backward slicing to determine the instructions that contribute to the value of the predicate. Then we perform symbolic execution along the slice to calculate a symbolic expression for each predicate. Based on that, our opaque predicate detector will construct formulas to represent the semantics of opaque predicates. We then solve them with a constraint solver. As discussed in Section 4.2, the detector conducts three rounds of scanning. The net result is a set of culprit branches corresponding to the opaque predicates.

4.3.1 Online Logging

As shown in Figure 4.4, LOOP's online part is built on a dynamic binary instrumentation framework. To undermine anti-malware detection, most malware developers apply different packers to compress or encrypt malware binaries. As a result, when a packed sample starts running, unpacking routines will first restore the original payload (e.g., decompress or decrypt) and then jump to the original entry point (OEP) to continue the execution. One of our implementation choices is we only detect opaque predicates within malware real payload. To this end, when a malware sample starts running, we first invoke our generic unpacking tool to monitor memory write operations. If a memory region pointed by the `eip` register is "written and then execute" [81], it indicates that we have identified the newly generated code. Then we activate our trace logging tool to start trace recording.

In general, analyzing all the instructions could be a tedious job. To keep the logged trace compact, LOOP supports on-demand logging to optionally record instructions of interest. In addition, our logging tool can perform dynamic taint tracking so that only the tainted instructions are collected.

4.3.2 Slicing and Symbolic Execution

Taking the logged trace as input, LOOP's offline analysis first lifts x86 instructions to BAP IL, which is a RISC-like intermediate language without side effect. In addition, the property of static single assignment (SSA) format facilitates tracing the use-def chain when we perform slicing. The symbolic execution is carried out on BAP IL as well.

Given a predicate (or branch), LOOP first identifies all the instructions that contribute to the calculation of this predicate. Note that x86 control transfer instructions typically depend on certain bits of the `eflags` register (e.g., `jz` and `jnz`). Therefore, the slicing criteria look like $\langle eip, zf \rangle$, where eip is the instruction pointer and zf is the abbreviation of the zero flag bit. Starting from the slicing criteria, we perform dynamic slicing [82] to backtrack a chain of instructions with data and control dependencies. We terminate our backward slicing when the source of the slice criteria satisfy one of the following conditions: constant values, static strings, user defined value (e.g., function return value) or input. Besides, we also

observe a case that the conditional logic is implemented without `eflags` register: `jecxz` jumps if register `ecx` is zero. LOOP handles this exception as well.

By labeling inputs or user defined values as symbols, we conduct symbolic execution along the slice to compute a symbolic expression for each predicate. The result will be passed to the opaque predicate detector, the core of LOOP's offline analysis. Figure 4.1 shows the components of our opaque predicate detector, which consists of three rounds of scanning to detect invariant, contextual and dynamic opaque predicates progressively. We will present each round of detection in the following subsections.

4.3.3 Invariant Opaque Predicates

Invariant opaque predicates refer to those predicates that are always true or false for all possible inputs. This kind of opaque predicates mainly relies on well-known algebraic theorems [21, 22]. Since they are easy to construct, invariant opaque predicates are the most frequently used ones. The detection method is straightforward by exploiting the invariant property. Tautology check in Figure 4.1 is used to prove whether the symbolic expression for each predicate in the trace always evaluates to true. For instance, one symbolic expression may be expressed as $\forall x \in Z.\ (x^3 - x)\%3 = 0$. We feed this formula to a constraint solver to prove its validity. Another characteristic of this category of predicates is that they are independent from each other, and therefore it is natural to parallelize the detection. We remove the identified invariant opaque predicates from the path condition to reduce its size. After that, the reduced path condition is sent to the second round of scanning.

4.3.4 Contextual Opaque Predicates

Different from the invariant property of the first category of predicates, a contextual opaque predicate relies on some program invariants so that it always produces the same value when certain precondition holds. At other places, such a predicate may evaluate to a different value. The precondition can be further obfuscated to camouflage the context information. Our detection method is based on the observation that a contextual opaque predicate (ψ_i) does not impose additional constraint on its prior path condition. Therefore, ψ_i should be *logically implied* by

its prior path condition: $\psi_1 \wedge \psi_2 ... \wedge \psi_{i-1}$; that is, $\psi_1 \wedge \psi_2 ... \wedge \psi_{i-1} \Rightarrow \psi_i$. The process of testing whether a predicate ψ_i and its preceding predicates satisfy the relation $\psi_1 \wedge \psi_2 ... \wedge \psi_{i-1} \Rightarrow \psi_i$ is called *implication check*. Without knowing which predicate is a potential contextual opaque predicate, we have to perform the implication check cumulatively starting from the first predicate to the last one. One challenge here is, subject to computing resources (e.g., memory and CPU), a complicated formula may be hard or infeasible to solve. To address this issue, we adopt a "short cut" strategy. When the generated formula becomes too complicated, we divide it into each single predicate and detect whether $\psi_j \Rightarrow \psi_i, j < i$. If this check passes, we do not need to prove $\psi_1 \wedge \psi_2 ... \wedge \psi_{i-1} \Rightarrow \psi_i$ any more, as the standard implication check will pass certainly.

The results of the second round of scanning are culprit branches that pass the implication check. Note that at this point, we cannot conclude whether these culprit branches are corresponding to real contextual opaque predicates. Recall the correlation property of dynamic opaque predicates discussed in Section 3.2, most dynamic opaque predicates (from the second to the last one in a family of related dynamic opaque predicates) satisfy the implication check as well. In the third round of detection, LOOP will distinguish these two categories and identify all correlated dynamic opaque predicates.

4.3.5 Dynamic Opaque Predicates

Dynamic opaque predicates consist of a family of *correlated* and *adjacent* predicates which produce the same boolean value in any given run. However, the value may switch to a different result in another run. Locating correlated predicates is the key to dynamic opaque predicate detection. A set of predicates are correlated means they are implied by each other, or in other words, they are *logically equivalent*. Therefore, predicates $\psi_1, \psi_2, ..., \psi_n$ are correlated iff

$$\psi_1 \Leftrightarrow \psi_2 \Leftrightarrow ... \Leftrightarrow \psi_n.$$

Assume we have two correlated predicates ψ_{i-1} and ψ_i. Recall that in the detection of contextual opaque predicates, we have proved that $\psi_{i-1} \Rightarrow \psi_i$. Now we have to go one step further to verify whether $\psi_i \Rightarrow \psi_{i-1}$. In logic, we have the following

Algorithm 1 Testing dynamic opaque predicates

```
    P: a set of predicates which pass the implication check
    T: the execution trace from which P is derived
 1: function TestDOP(T, P)
 2:     for each ψ_i in P do
 3:         T' ← GenTrace(¬ψ_{i−1})
 4:         ψ'_i ← Next(T', ¬ψ_{i−1})
 5:         if ψ'_i ≠ ¬ψ_i then
 6:             return False
 7:         end if
 8:         if ¬ψ_{i−1} ⇒ ¬ψ_i then
 9:             if T_{i−1..i} ≈ T'_{i−1..i} then
                    // semantically equivalent
10:                 return True
11:             else
12:                 return False
13:             end if
14:         else
15:             return False
16:         end if
17:     end for
18: end function
```

equation.

$$\psi_i \Rightarrow \psi_{i-1} \equiv \neg\psi_{i-1} \Rightarrow \neg\psi_i$$

Therefore, we alternatively verify whether $\neg\psi_{i-1}$ implies $\neg\psi_i$ by generating a new input. Another property of dynamic opaque predicates is that they utilize code clone to make sure the same instructions are executed in any case. As a result, we have to compare the two traces to ensure they are semantically equivalent. Similar to the trace-oriented binary diffing tool [36], our comparison approach relies on symbolic execution and theorem proving techniques so that we can find that two code pairs with ostensibly different syntax are semantically equivalent.

Suppose ψ_{i-1} and ψ_i are dynamic opaque predicates and ψ_i has been identified as a culprit branch satisfying implication check. The procedure of our third round of detection is shown in Algorithm 1. First, we negate the predicate ψ_{i-1} and use constraint solver to generate a new input to follow the path $\psi_1 \wedge \psi_2 ... \wedge \neg\psi_{i-1}$. Given the new trace, the predicate following $\neg\psi_{i-1}$ is ψ'_i. Then we verify ψ'_i is equivalent to $\neg\psi_i$. Next, we test whether $\neg\psi_{i-1} \Rightarrow \neg\psi_i$. If it succeeds, we will

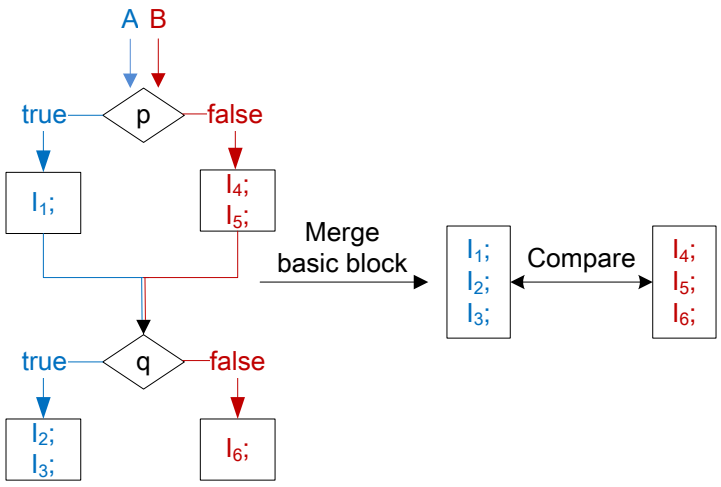


Figure 4.5: Trace segment comparison (A and B represent different inputs).

compare the trace segments controlled by $\psi_{i-1} \wedge \psi_i$ and $\neg\psi_{i-1} \wedge \neg\psi_i$. Different from previous semantics-based binary diffing tools [36], whose comparison unit is a single basic block, LOOP merges all the basic blocks which have control flow dependence with $\psi_{i-1} \wedge \psi_i$ or $\neg\psi_{i-1} \wedge \neg\psi_i$ as a trace segment (as shown in Figure 4.5). LOOP carries out symbolic execution for the trace segment and represents its input-output relation with a set of formulas. At last, LOOP uses a constraint solver to compare the output formula pairs and then finds a bijective equivalent mapping among them. If so, we conclude that these two trace segments are semantically equivalent and therefore ψ_{i-1} and ψ_i are dynamic opaque predicates. If $\neg\psi_{i-1} \Rightarrow \neg\psi_i$ fails or new trace segment is different to the original one, we conclude ψ_i is a contextual opaque predicate.

Following the similar idea, we can identify the correlated predicates containing more than two predicates and then check the equivalence of the trace segments which have control dependence on these predicates.

4.3.6 Solving Opaque Predicate Formulas

It is evident that the efficacy of LOOP depends on the capability of constraint solvers in proving the validity of a formula. State-of-the-art constraint solvers (e.g., STP [83], Z3 [84]) have supported all arithmetic operators found in the C programming language (e.g., bitwise operations, multiplication, division, and modular arithmetics). Table 4.1 shows ten mathematical formulas that are commonly used

Table 4.1: Common mathematical formulas and their STP solving time.

Formulas	Solving time (s)
$\forall x \in Z.\ x^2 \geq 0$	0.003
$\forall x \in Z.\ 2 \mid x(x+1)$	0.008
$\forall x \in Z.\ 3 \mid x(x+1)(x+2)$	0.702
$\forall x, y \in Z.\ 7y^2 - 1 \neq x^2$	0.008
$\forall x \in Z.\ (x^2+1)\%7 \neq 0$	17.762
$\forall x \in Z.\ (x^2+x+7)\%81 \neq 0$	22.657
$\forall x \in Z.\ (4x^2+4)\%19 \neq 0$	15.392
$\forall x \in Z.\ 4 \mid x^2(x+1)(x+1)$	0.012
$\forall x \in Z.\ 2 \mid x \vee 8 \mid (x^2-1)$	0.022
$\forall x \in Z.\ 2 \mid \lfloor \frac{x^2}{2} \rfloor$	0.015

as invariant opaque predicates, including integer division, modulo and remainder operations. We solve these formulas with STP in our testbed and present the solving time in the second column. Most formulas only need less than 0.1 seconds, which are almost negligible. However, we notice that formulas involving modular arithmetic increase the solving time considerably. For example, the solving time for three complicated modular arithmetic formulas in Table 4.1 varies from 15 to 23 seconds. Even so, compared with other methods which need to run a large set of test inputs (e.g., statistical analysis [1] and fuzz testing [31]), our approach achieves better performance and accuracy.

4.3.7 Whole Program Deobfuscation

After three rounds of detection, LOOP returns all possible opaque predicates in an execution trace. Our result can be fed back to security analysts to further deobfuscate the cluttered control flow graph of a whole program. For example, the unreachable paths introduced by invariant opaque predicates are discarded; the redundant contextual opaque predicates are cut off as well. To reverse the effect of dynamic opaque predicates, we remove the set of correlated dynamic opaque predicates and replace the corresponding multiple paths with a single straight-line path. Since our approach is trace-oriented, we deobfuscate a part of the control flow graph each time. To increase path coverage, we can leverage automatic input generation techniques [62, 65].

4.4 Implementation

LOOP's online logging part consists of two tools, generic unpacking and trace logging, which are implemented based on the Pin DBI framework [85] (version 2.12) with 1,752 lines of code in C/C++. To make the logged trace compact, we start trace logging when unpacking routine finishes and support on-demand logging for instructions of interest. LOOP's offline analysis part is implemented on top of BAP [37] (version 0.8) with 2,588 lines of OCaml code. We rely on BAP to lift up x86 instructions to the BAP IL and convert BAP IL to CVC formulas. LOOP's backward slicing is performed on BAP IL; the opaque predicate detector and trace segment comparison tool are built on BAP's symbolic execution engine. We use STP [83] as our constraint solver. Besides, we write 644 lines of Perl scripts to glue all components together to automate the detection. To facilitate future research, we have made LOOP code available at `https://github.com/s3team/loop`.

4.5 Evaluation

We evaluate LOOP's effectiveness to automatically detect various opaque predicates. We test LOOP with a set of Linux common utilities and highly obfuscated malicious programs. We make sure we have the ground truth so that we can accurately assess false positives and false negatives. Also, since LOOP is strongly motivated by its application, we evaluate the impact of LOOP in the task of code normalization for metamorphic malware. The experiments are performed on a machine with a Intel Core i7-3770 processor (Quad Core, 3.40GHz) and 8GB memory, running both Ubuntu 12.04 and Windows XP SP3.

4.5.1 Evaluation with Common Utilities

This experiment is designed to evaluate the effectiveness and efficiency of our method. First, we select ten widely used utility programs in Linux as our test cases. To ensure the samples' variety, those candidates are picked from different areas, including data compression, core utilities, regular expression search, hash computing, web file transfer, and HTTP server. Since all of those programs are open source, we can easily verify our detection results. We implement automatic

Table 4.2: Contextual opaque predicates used in common utilities evaluation.

1	$\forall x \in Z.\ x > 5 \Rightarrow x > 0$
2	$\forall x \in Z.\ x > 3 \Rightarrow x^2 - 4x + 3 > 0$
3	$\forall x \in Z.\ x\%4 = 0 \Rightarrow x\%2 = 0$
4	$\forall x \in Z.\ x\%9 = 0 \Rightarrow x\%3 = 0$
5	$\forall x \in Z.\ x\%10 = 0 \Rightarrow x\%5 = 0$
6	$\forall x \in Z.\ 3 \mid (7x - 5) \Rightarrow 9 \mid (28x^2 - 13x - 5)$
7	$\forall x \in Z.\ 5 \mid (2x - 1) \Rightarrow 25 \mid (14x^2 - 19x - 19)$
8	$\forall x, y, z \in Z.\ (2 \nmid x \wedge 2 \nmid y) \Rightarrow x^2 + y^2 \neq z^2$

Table 4.3: Evaluation results on Linux common utilities.

Program	#Predicates	Invariant			Contextual			Dynamic			(#FP, #FN)
	(no-taint, taint)	#OP	SE (s)	STP (s)	#OP	SE (s)	STP (s)	#OP	SE (s)	STP (s)	
bzip	(6,313, 13)	4	0.9	1.4	6	1.2	2.3/1.9	2	1.8	2.4	(5, 0)
grep	(4,969, 16)	5	0.9	1.5	6	1.8	1.6/1.3	3	2.4	0.3	(6, 0)
ls	(5,867, 21)	13	3.3	3.9	3	22.4	3.5/2.8	1	12.5	2.1	(10, 0)
head	(1,496, 11)	6	0.7	2.5	7	1.1	1.3/1.3	1	1.9	1.0	(1, 0)
md5sum	(3,450, 14)	3	2.8	4.5	3	1.5	25.0/20.3	1	1.0	2.2	(0, 0)
thttpd	(6,605, 124)	3	8.7	16.8	3	3.6	5.4/2.2	1	1.4	0.7	(0, 0)
boa	(4,718, 131)	3	6.8	18.7	3	2.5	6.2/1.8	1	1.7	0.8	(0, 0)
wget	(3,230, 36)	5	3.0	7.6	3	2.0	2.2/1.2	1	1.5	0.7	(2, 0)
scp	(2,402, 30)	5	3.4	5.8	4	2.8	4.1/2.4	1	1.5	0.7	(3, 0)
libpng	(25,377, 446)	7	51.4	351.6	4	14.6	33.2/11.6	2	10.4	5.2	(6, 0)

opaque predicate insertion as an LLVM pass, based on Obfuscator-LLVM [25]. For each program, we insert seven opaque predicates, including three invariant, three contextual and one dynamic opaque predicates. The three invariant opaque predicates are randomly selected from Table 4.1; the three contextual opaque predicates are chosen from Table 4.2. We use the example of dynamic opaque predicates shown in Figure 4.2. At the same time, we make sure that all the opaque predicates can be reached by the test inputs.

We first label the inputs of test cases as tainted and record tainted instructions. Then we run LOOP's offline analysis to detect the opaque predicates. Table 4.3 shows the experimental results. The second column shows the number of predicates before and after taint. It is evident that forward taint propagation reduces the number of predicates significantly. For each category of opaque predicate, we report the number of opaque predicate detected (#OP), the time of symbolic execution (SE) and the time of running STP solver (STP). Note that in the column of STP time for contextual opaque predicates, we list the different time before and after "short cut" optimization (see Section 4.3.4).

For the majority of our test cases, the symbolic execution and STP solver only take several seconds. Because libpng's trace size is large and its invariant opaque predicates inserted involve modulus arithmetic, the corresponding STP solving time is the longest. The data presented in the eighth column indicate that the effect of "short cut" optimization is encouraging, especially for the cases with large path formulas (e.g., libpng). Furthermore, we manually verify each logged traces and find that our approach successfully diagnoses all the opaque predicates if there is any; that is, we have zero false negatives (#FN in the last column).

To test false positives of our approach, namely whether LOOP mistakes a normal condition as an opaque predicate, we conduct a similar evaluation on our test cases' clean version (no opaque predicate insertion). Contrary to our expectation, we notice that seven out of ten cases detect opaque predicates but all of them are false positives (#FP in the last column). We look into the factors leading to the false positives and find that one major reason is "under tainting" [71], which is a common problem in taint analysis. Generally, under tainting means instructions that should be tainted are not recorded. As a result, under tainting will mistakenly replace some symbols with concrete values. For instance, supposing `y` with a concrete value of `2` is not labeled as a symbol in the predicate `y > 1`, the predicate in the trace would be `2 > 1`, which is a tautology, and LOOP will issue a false alarm.

4.5.2 Evaluation with Obfuscated Malware

Opaque predicates are also widely used by malware developers. Moreover, opaque predicates are typically integrated with other obfuscation methods to impede reverse engineering attempts. To evaluate the resilience of LOOP against various obfuscation methods, we collect 15 malware binaries from VX Heavens.[1] These malware samples are chosen for two reasons: 1) they are representative in obfuscation techniques; 2) we have either source code (e.g., QQThief and KeyLogger) or detailed reverse engineering reports (e.g., Bagle and Mydoom). Hence, we can accurately evaluate our detection results.

Table 4.4 shows a variety of obfuscation techniques with different purposes. Column $4 \sim 9$ represent the methods to obfuscate code and data, such as binary compression and encryption packers, junk code, code reorder, and opaque constant.

[1] http://vxheaven.org/src.php

Table 4.4: Various obfuscation methods applied by malicious program.

Sample	Type	Size (kb)	Packer	Compres. packer	Encrypt. packer	Junk code	Code reorder	Opaque constant	Call-stack tampe.	CFG flatten	Obfuscat. transf.	Anti-debugging	Anti-VM	# OP	(#FP, #FN)	Time (s)
Bube	Virus	12	FSG	✓		✓	✓	✓	✓					(0,0,0)	(0, 0)	3.4
Tefuss	Virus	29	UPX	✓		✓	✓		✓			✓	✓	(2,0,0)	(0, 0)	4.2
Champ	Virus	13	PECompact	✓				✓	✓			✓	✓	(0,0,0)	(0, 0)	3.0
BullMoose	Trojan	30	VMProtect		✓	✓	✓					✓		(5,3,1)	(2, 0)	7.8
QQThief	Trojan	32	Yoda's Protector	✓	✓			✓	✓		✓	✓		(3,0,0)	(1, 0)	15.4
KeyLogger	Trojan	33	ASPack	✓	✓				✓		✓			(8,1,1)	(0, 0)	22.3
Autocrat	Backdoor	276	PECompact	✓				✓	✓			✓	✓	(12,0,0)	(4, 0)	244.5
Codbot	Backdoor	30	ASPack	✓	✓				✓		✓			(2,0,0)	(0, 0)	8.2
Loony	Backdoor	20	ASPack	✓	✓				✓		✓			(0,0,0)	(1, 0)	5.3
Branko	Worm	17	Themida		✓					✓	✓	✓	✓	(10,0,0)	(0, 0)	8.5
Hunatcha	Worm	61	PolyEnE		✓	✓		✓		✓	✓			(90,3,1)	(0, 0)	36.2
Bagle	Worm	47	UPack	✓	✓	✓	✓							(6,0,0)	(2, 0)	24.6
Sasser	Worm	60	UPack	✓	✓	✓	✓					✓		(4,0,0)	(0, 0)	18.2
Mydoom	Worm	41	ASProtect	✓	✓	✓	✓	✓			✓			(6,0,0)	(1, 0)	132.3
Zeynep	Worm	85	Yoda's Protector	✓	✓			✓	✓		✓	✓		(8,1,0)	(3, 0)	192.3

```
call near ptr GetFontData ; eax = 0xFFFFFFFF
                          ; ebp, esp are even numbers
and eax, ebp              ; eax = eax & ebp
                          ; eax is an even number
inc eax                   ; eax += 1, eax is an odd number
and eax, 01h
cmp eax, 0                ; eax = 01h
jne   new_target          ; always true
```

Figure 4.6: Example of an opaque predicate in malware.

Column 10 $\sim$ 12 denote the control flow obfuscation methods in addition to opaque predicates, including call-stack tampering, CFG flatten, and obfuscated control transfer target. The methods in column 13 and 14 are used to detect the debugging and virtual machine (VM) environment. The "# OP" column presents the number of opaque predicates detected by LOOP. The triple such as (5,3,1) represents the number of invariant, contextual, and dynamic opaque predicates, respectively. We find that most malware samples (12 out of 15) are embedded with opaque predicates and invariant opaque predicates are the most frequently used. The high number (90) of invariant opaque predicates detected in Hunatcha is caused by loop unrolling. With the help of source code and reverse engineering reports, we count false positives and false negatives (shown in column 16). Similar with common utilities' results, LOOP achieves zero false negatives. That is, LOOP does not

Table 4.5: Speed up metamorphic malware variants matching

Family	Basic blocks reduction (%)	Isomorphism speedup (X)
Metaphor	26	2.0
Lexotan32	20	1.6
Win32.Evol	16	1.2

miss any opaque predicates. The last column lists the total offline analysis time. Note that our generic unpacking cannot handle virtualization obfuscators [86] such as VMProtect[2] and Themida[3]. In our test cases, we find two malware samples (BullMoose and Branko) are obfuscated with virtualization obfuscation. As a result, the logged trace mixes the code of virtualization interpreter with the code of malicious payload. LOOP nevertheless detects opaque predicates successfully.

Figure 4.6 shows an opaque predicate we detected in KeyLogger Trojan. This opaque predicate utilizes the fact of stack memory alignment, and therefore both `ebp` and `esp` are even numbers. After several arithmetic operations, the last branch is always true. In summary, our experiments show that LOOP is effective in detecting opaque predicates in obfuscated binary code with a zero false negative rate. Considering that LOOP aims to provide a general and automatic deobfuscation solution, which usually involves tedious manual work, the false positive rate is tolerable.

4.5.3 Metamorphic Malware Matching

To confirm the value of our approach in malware defenses, we also test LOOP in the task of code normalization for metamorphic malware [19,20]. Metamorphic malware mutates its code during infection so that each variant bears little resemblance to another one in syntax. It is well known that metamorphism can undermine the signature-based anti-malware solutions [28]. Bruschi et al. [19, 20] propose code normalization to reverse the mutation process. To test whether an instance of metamorphic malware is present inside an infected host program, they compare malicious code and normalized program by inter-procedural control flow subgraph isomorphism. The drawback is that they do not handle opaque predicates, although

[2]http://vmpsoft.com/
[3]http://www.oreans.com/themida.php

opaque predicates are one of the commonly used mutation methods. Opaque predicates can seriously thwart normalized control flow graph comparison [19]. We re-implement their code normalization based on BAP and test the speedup of normalized control flow graph isomorphism under the preprocess of LOOP on three famous metamorphic malware families.

Since the three metamorphic malware samples are all file-infecting, we first force each malware to infect 20 Cygwin utilities.[4] For each family, we follow similar steps as in Bruschi et al. to normalize infected programs and compare their control flow graphs (CFG) with malicious code's CFG, leveraging VFLIB library [87]. In addition, we apply LOOP to preprocess infected programs to remove the corresponding superfluous branches and infeasible paths. Table 4.5 shows the improvements introduced by our approach on average. Compared with the results without applying LOOP, we remove redundant basic blocks as much as 26% and speed up subgraph isomorphism by a factor of up to 2.0 (e.g., Metaphor).

4.6 Discussions and Future Work

We further discuss about our design choices, limitations and future work in this section.

4.6.1 Dynamic Approach

Our approach bears the similar limitations as dynamic analysis in general. For example, LOOP can only detect opaque predicates executed at run time. Static analysis might explore all the possible paths in the program. However, even static disassembly of stripped binaries is still a challenge [88, 89]. Moreover, the various obfuscation techniques listed in Table 4.4 will undoubtedly deter extracting accurate control flow graph from binary code. We believe our approach, based on the test cases that execute opaque predicates, is practical in analyzing a malicious program. A possible way to increase path coverage is to leverage test-generation techniques [62, 65] to automatically explore new paths. Another concern we want to discuss is scalability issue. The size of a slice may become significant for a program with high workload, and our detection approach is linearly dependent on the size of

[4] www.cygwin.com

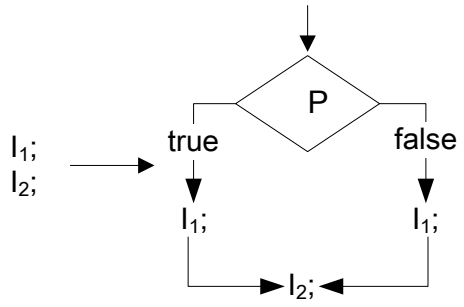


Figure 4.7: Example of two-way opaque predicates.

a slice. In that case, LOOP has to analyze a large number of predicates, resulting in a substantial performance slowdown. One way to alleviate the high overhead is to detect opaque predicates in parallel. We plan to explore this direction in our future work. Detecting repetitive opaque predicates due to loop unrolling leads to performance penalty. We will extend our tool with loop identification in the next version.

4.6.2 Floating Point

The effect of LOOP is restricted by the capability of the constraint solver and symbolic execution tools. One example is floating point. Since precisely defining the rounding semantics of floating point is challenging, current binary symbolic execution does not support floating point instructions (e.g., fdiv and fmul) [37]. As a result, currently LOOP is unable to detect opaque predicates involving floating point operations. One might argue that attackers can easily get around LOOP by using floating point equations. However, using floating point equations in malware increases the possibility of detection as well, because both floating point instructions and related numerics library API calls (e.g., log, exp and sqrt) are rarely seen in malicious code.

4.6.3 Two-way Opaque Predicates

Figure 4.7 shows a subtle case of opaque predicate, in which two possible directions will sometimes be taken and $I_1; I_2$ are executed in any case. P does not belong to the categories we summarized in Section 3.2 and therefore we cannot ensure P

is an opaque predicate in a single trace. One way to detect such case is to check semantic equivalence of P's two jump targets [90].

Chapter 5
Generalized Dynamic Opaque Predicates

5.1 Introduction

In this chapter, we present a systematic design of a new control flow obfuscation method, *Generalized Dynamic Opaque Predicates*, which is able to inject diversified dynamic opaque predicates into complicated program structures such as branch and loop. Being compared with the previous technique which can only insert dynamic opaque predicates into straight-line program, our new method is more resilient to program analysis tools. We have implemented a prototype tool based on the LLVM compiler infrastructure [38]. The tool first performs fine-grained data flow analysis to search possible insertion locations. After that it automatically transforms common program structures to construct dynamic opaque predicates. We have tested and evaluated the tool by obfuscating several hot functions of GNU core utilities with different obfuscation levels. The experimental results show that our method is effective and general in control flow obfuscation. Besides, we demonstrate that our obfuscation can defeat the commercial binary difference analysis tools and the state-of-the-art formal program semantics-based deobfuscation methods. The performance data indicate that our proposed obfuscation only introduces negligible overhead.

In summary, we make the following contributions.

The work of this chapter is published in the 19th Information Security Conference [91].

- First, we propose an effective and generalized opaque predicate obfuscation method. Our method outperforms existing work by automatically inserting opaque predicates into more general program structures like branches and loops, whereas previous work can only work on straight-line code.
- Second, we demonstrate our obfuscation is very resilient to the state-of-art opaque predicate detection tool.
- Third, we have implemented our method on top of LLVM and the source code is available.

The rest of this chapter is organized as follows. Section 5.2 presents our new obfuscation method, generalized dynamic opaque predicates in detail. Section 5.3 presents our implementation details. We evaluate our method in Section 5.4.

5.2 Method

In this section, we present the details of the generalized dynamic opaque predicates method. First, we introduce the concept of correlated predicate. After that, we explain how to insert generalized dynamic opaque predicates into straight-line programs, branches and loops.

5.2.1 Correlated Predicates

Correlated predicate, as briefly discussed in Section 3.2, is a basic concept in dynamic opaque predicate. In this section, we present the formal definition of correlated predicate. First we need to define *correlated variables*. *Correlated variables* is a set of variables that are always evaluated to the same value in any program execution. One common example of correlated variables is the aliases of the same variable, like the pointers in C or the references in C++ or Java.

Correlated predicates are a set of predicates that are composed of correlated variables and have a fixed relation of their true value. The fixed relation means that, given a set of correlated predicates, if the true value of one of them is given, all other predicates' true value are known. Usually, it is intuitive to construct correlated predicates using correlated variables. Table 5.1 shows some examples of correlated predicates. The integer variables `x`, `y` and `z` in the first column is

Table 5.1: Examples of correlates predicates.

CV	CP_1	CP_2	CP_3
x	`x > 0`	`x%2 == 1`	`x+x > 0`
y	`y > 0`	`y%2 == 0`	`2*y <= 0`
z	`z <= 0`	`z%2 == 1`	`z<<1 > 0`

the correlated variables (CV). The CP_1, CP_2 and CP_3 columns show three sets of different correlated predicates.

Here we take the CP_2 column as an example to show how correlated predicates work. First, since `x`, `y` and `z` are correlated integer variables, they are always equivalent. There are three predicates in CP_2, `x%2 == 1`, `y%2 == 0` and `z%2 == 1`. Note that `x`, `y` and `z` are integer variables, so they are either even or odd. Therefore, given the true value of any one of these predicates, we can immediately get the others' true values. Furthermore, it is not necessary that correlated predicates have similar syntax form. We can use semantically equivalent operations to create correlated predicates. CP_3 shows such an example. Although the syntax of each predicate is different from others, they still meet the definition of correlated predicates.

One problem we need to pay attention to is that the value of the correlated variables should not be changed during the dynamic opaque predicates, which ensures that every correlated variable are evaluated to the same value in all dynamic opaque predicates in one execution. Therefore, we compute the def-use chain inside a function and choose the section between two definitions of a variable as the candidate to be obfuscated. Note that pointer access operations could still cause the variable's value changes. Our solution is performing a simple alias analysis to decide whether the pointer is an alias of the variable. If not, we can include the pointer access instructions inside the dynamic opaque predicates; otherwise not. Since alias analysis is complicated and difficult, we only run a light-weighted address-taken algorithm [92] in our implementation. It is flow-insensitive and context-insensitive. If the analysis cannot tell whether the pointer is an alias of the correlated variable, we will conservatively consider that it could point to the variable and exclude it from the dynamic opaque predicates candidates.

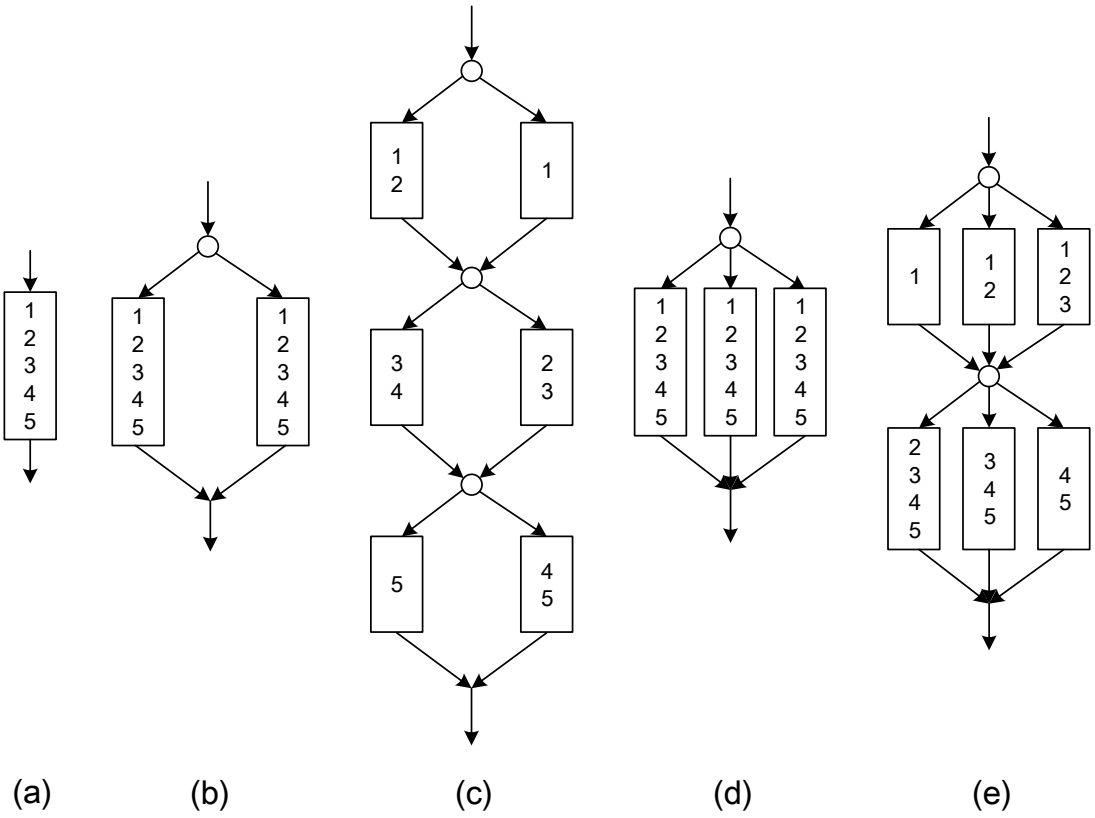


Figure 5.1: Dynamic opaque predicate insertion in straight-line code.

5.2.2 Straight-line Code

In this section, we present how to insert dynamic opaque predicate into a straight-line code. Before digging into the details, we first explain the symbols in the figures as follows.

1. A *rectangle* is a basic block.
2. A *number* in a rectangle represents one instruction.
3. A *circle* indicates a correlated predicate.
4. An *arrow* between two basic blocks indicates the control flow transfer. Typically, it is a conditional or unconditional jump. If there is only one arrow between two blocks, it is an unconditional jump; otherwise, it is a conditional jump.

Given the definition above, Fig. 5.1(a) shows a straight-line code which contains only one basic block, in which there are five sequential instructions. If a straight-line code comprises multiple basic blocks which are connected by unconditional jumps, it can be merged into one basic block. So for the ease of understanding, we use the single basic block example to present straight-line code.

When inserting dynamic opaque predicates into straight-line code, we have two strategies, depth-first and breadth-first, whose obfuscation result is shown in Fig. 5.1(c) and Fig. 5.1(e). Here we introduce the depth-first style first and briefly discuss the breadth-first later since they are similar. When inserting dynamic opaque predicates in depth-first style, we select the first correlated predicate and then make a copy of the original basic block, as shown in Fig. 5.1(b). After that, the two basic blocks are split at different locations so as to create two chains of basic blocks in which each basic block are different with each other. At last, we insert other correlated predicates to ensure that the control flow takes either all left branches or all right branches.

Furthermore, when inserting depth-first dynamic opaque predicates, we could insert as many correlated predicates as we can by splitting the basic blocks at different locations. As shown in Fig. 5.1(c), those basic blocks constitute two chains, in which the execution flow will either take every left branch or right branch. We call the basic block sequence that consists of all left or right branches an *opaque trace*. In this paper, the multiple execution traces caused by the effect of opaque predicates are called *opaque trace*. As shown in Fig. 5.1(c), if the execution flow takes all the left branches, the opaque trace is `[1 2]->[3 4]->[5]`. Similarly, when taking all right branches, the opaque trace is `[1]->[2 3]->[4 5]`. Therefore, Fig. 5.1(c) contains two opaque traces.

Generally speaking, the steps to insert depth-first dynamic opaque predicates to a single basic block BB are described as follows.

1. Select a correlated variable and creating the first correlated predicate accordingly.

2. Clone a new basic block BB' from BB.

3. Split BB and BB' at different locations to create two sequences of basic blocks, or say, two opaque traces T_1 and T_2:

$$T_1 = BB_1 \rightarrow BB_2 \rightarrow \cdots \rightarrow BB_n$$
$$T_2 = BB'_1 \rightarrow BB'_2 \rightarrow \cdots \rightarrow BB'_n$$

4. Create and insert the remaining $n - 1$ correlated predicates.

5. Insert conditional or unconditional jumps into the end of each basic block to create the correct control flow.

The other strategy is breadth-first inserting dynamic opaque predicates. It create more opaque traces via correlated predicates that have multiple branches. The inserting process is similar as depth-first. Assuming each predicate has three branches, the first step is to select and insert the first correlated predicate and create two copies of the original basic block as shown in Fig. 5.1(d). Then split the three basic blocks at different offsets so as to create three opaque traces. At last, insert the other correlated predicates and other jump instructions to adjust the CFG. The result is shown in Fig. 5.1(e).

Furthermore, we can easily create more complicated generalized dynamic opaque predicates by iteratively applying depth first and breadth first injection. For example, the basic block `[1,2,3]` can also be split to create a depth first generalized dynamic opaque predicate. Note that it naturally breaks the adjacency of the two predicates in Fig. 5.1(e). Being compared with the conventional dynamic opaque predicate shown in Section 3.2 which only has two adjacent predicates p and q, our method can insert more generalized and non-adjacent dynamic opaque predicates in straight-line code.

5.2.3 Branches

In the previous section, we present the approach to inserting dynamic opaque predicates into straight-line code. However, real world programs also consist of other structures such as branches and loops. When considering inserting dynamic opaque predicates into branches or loops, one straight forward idea is only inserting dynamic opaque predicates into basic blocks independently by treating them as

straight-line code. However, this idea has one obvious problem: it doesn't spread the dynamic opaque predicates across the branch or loop condition, so essentially it is still the same as what we have done in Section 5.2.2.

In this section, we describe the process to insert dynamic opaque predicates into a branch program, which improves the program obfuscation level. For the ease of presenting our approach, we consider the branch program which contains three basic blocks as shown in Fig. 5.2(a). Our solution can also be applied to more complicated cases such as each branch contains multiple basic blocks. As shown in Fig. 5.2, `Cond` is the branch condition. BB_1 is located before the branch condition. BB_2 is the true branch and BB_3 is the false branch.

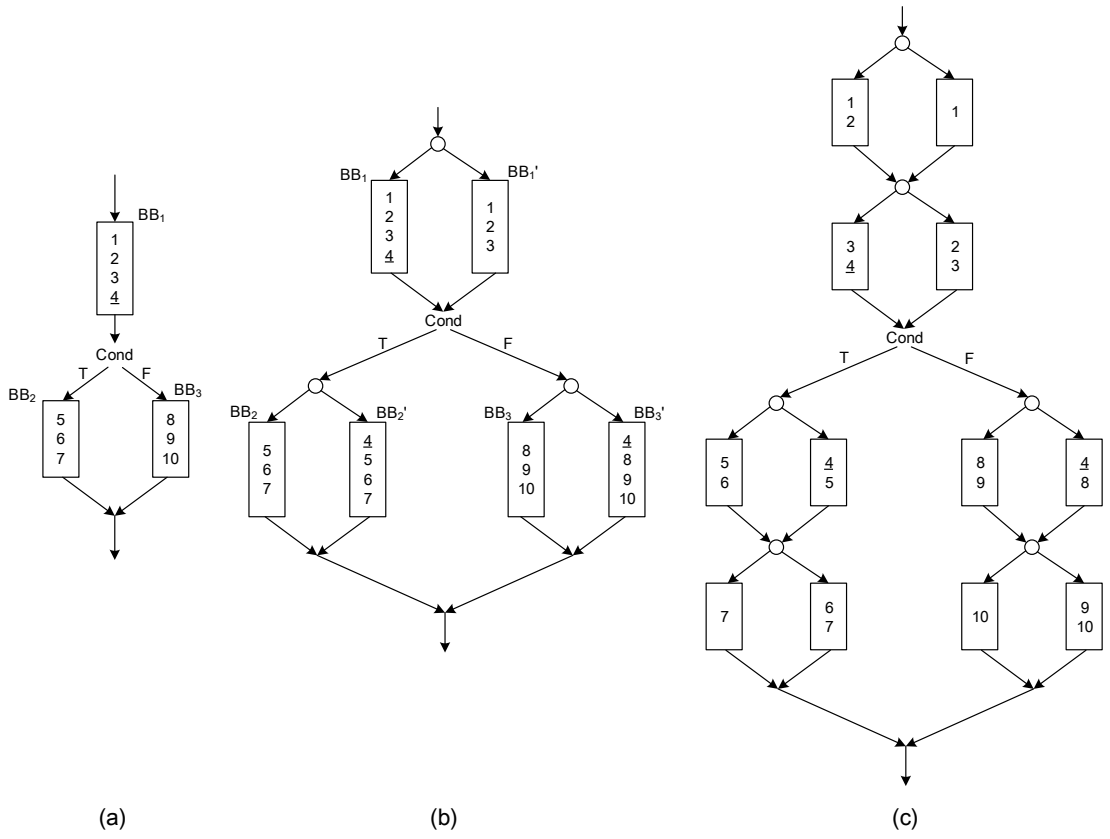


Figure 5.2: Dynamic opaque predicate insertion in a branch program.

As the first step of inserting branch dynamic opaque predicate, we backwards search for an instruction that is independent from all instructions until the branch condition, and also independent from the branch instruction. In this paper, this instruction is called a *branch independent instruction*. Essentially, it can be moved

across the branch condition so as to create the offset in different opaque traces. In Fig. 5.2(a), the underlined instruction 4 is a branch independent instruction. Based on our observation, there are plenty of branch independent instructions. For example, the Coreutils program `ls` contains 289 branch conditions, in each of which we find at least one branch independent instruction. Typically, these instructions prepare data which are used both in the true and false branch.

After identifying the branch independent instruction, we select and insert the correlated variables, then make a copy of each basic blocks. Moreover, we move the instruction 4 along the right opaque trace across the branch condition and Fig. 5.2(b) shows the result. Note that due to instruction 4 is branch independent, so moving it to the head of basic blocks in the branches will not change the original program's semantics. At last, we create straight-line code dynamic opaque predicates for BB_1, BB_2 and BB_3. The final result of the obfuscated CFG is shown in Fig. 5.2(c). We briefly summarize the steps of inserting dynamic opaque predicates into a branch program as follows.

1. Find the branch independent instruction in BB_1.
2. Select and insert the correlated predicates.
3. Clone BB_1, BB_2 and BB_3 as BB'_1, BB'_2 and BB'_3.
4. Move the branch independent instruction from BB'_1 to BB'_2 and BB'_3.
5. Split basic blocks and create dynamic opaque predicates as in straight-line code.

5.2.4 Loops

Previous sections present the details about how to insert dynamic opaque predicates into straight-line code and branch programs. In this section, we consider inserting dynamic opaque predicates into a loop. In this paper, a loop refers to a program which contains a backward control flow, such as Fig. 5.3(a). BB_1 is the first basic block of the loop body and BB_2 is the last one. The dashed line indicates other instructions in the basic block. The dashed arrow means other instructions in the loop body, which could be a basic block, branch or even another loop. Particularly,

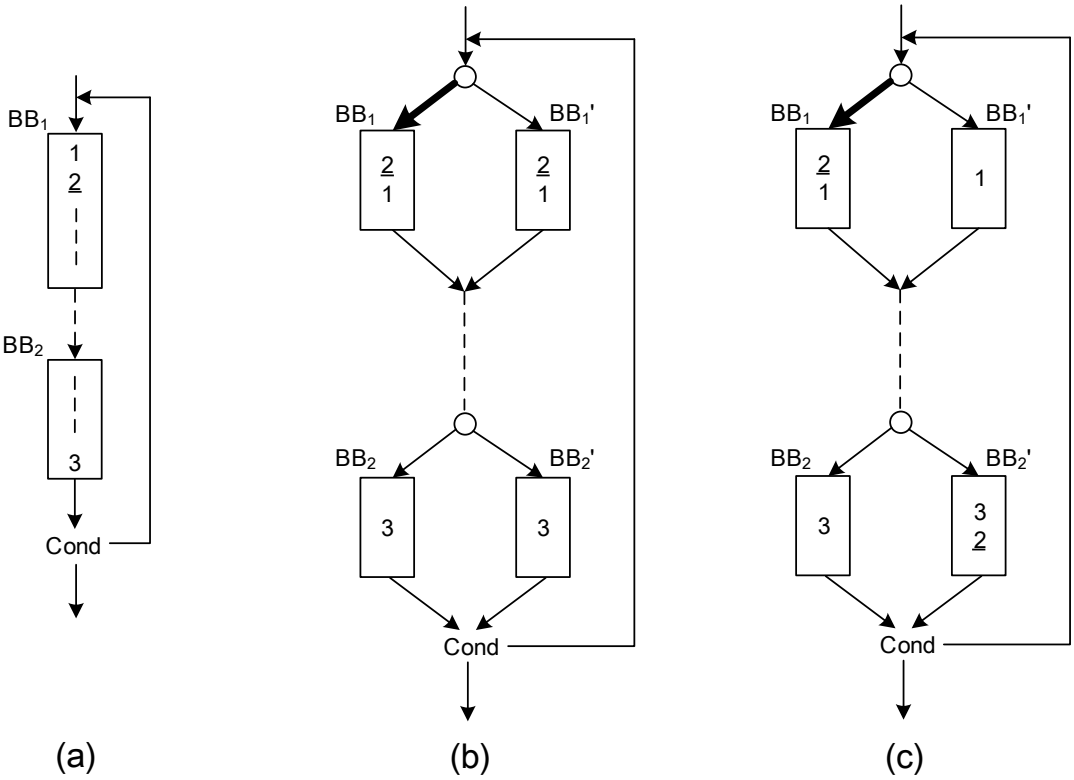


Figure 5.3: Dynamic opaque predicate insertion in a loop.

if there is only one basic block in the loop body, BB_1 and BB_2 refer to the same basic block.

The key idea in inserting dynamic opaque predicates to a loop program is finding a *loop independent instruction* and moving it across the loop condition in the same opaque trace. We define *loop independent instruction* as an instruction in a loop whose operands are all loop invariants. Loop invariant is a classical concept in compiler optimization. A variable is called loop invariant if its value never changes no matter how many times the loop is executed. For instance, Fig. 5.4 shows a loop invariant. The variable `m` is defined outside the loop and is never changed inside the loop. Each iteration of the loop accesses the same array element `A[m]` and assigns it to the variable `x`. Therefore, `m`, `A[m]` and `x` are loop invariant. Note that here we use the C source code to present the idea. Actually we are working on the compiler IR level, where every instruction is close to a machine instruction. As a result, in the IR level, all instructions that only operate the loop invariants are loop independent instructions. For instance, the instruction that load the value of `A[m]`

```
1 for (i = 0; i < 10; i++) {
2     x = A[m];    /* loop invariant */
3     B[i] = x * i;
4 }
```

Figure 5.4: An example of loop invariants.

from memory to `x` is an loop independent instruction. Based on our observation, there are plenty of loop independent instructions inside a loop body, such as the instructions to compute a variable's offset address. In the experiment, we find at least loop independent instruction for each of the 61 loops in the Coreutils program `ls`.

In traditional compiler optimization, the loop independent instructions are extracted out of the loop body so as to reduce the loop body size and further improve the runtime performance. All compiler frameworks implement a data flow analysis to analyze and identify the loop invariants. In this paper, we take advantage of the loop independent instructions to create the offset between opaque traces. Consider the example shown in Fig. 5.3(a). First, we search and identify that instruction 2 is a loop independent instruction. Second, we lift the instruction 2 to the beginning of the loop body, since other instructions might need the output of instruction 2. Then we make copies of BB_1 and BB_2 as BB'_1 and BB'_2. After that we select the correlated predicates and initialize the first one to ensure that it takes the left branch. The bold arrow in Fig. 5.3(b) indicates the initialized predicates. We will soon discuss the reason. At last, the loop independent instruction 2 is moved from BB'_1 to BB'_2 and the final result is shown in Fig. 5.3(c). We summarize the steps of creating loop dynamic opaque predicates as follows.

1. Find the loop independent instruction I_i.
2. Lift I_i to the beginning of the loop body in BB_1.
3. Select the correlated predicates and initialize the first one correctly.
4. Clone BB_1 and BB_2 as BB'_1 and BB'_2.
5. Remove I'_i from BB'_1 and add it to the end of BB'_2.

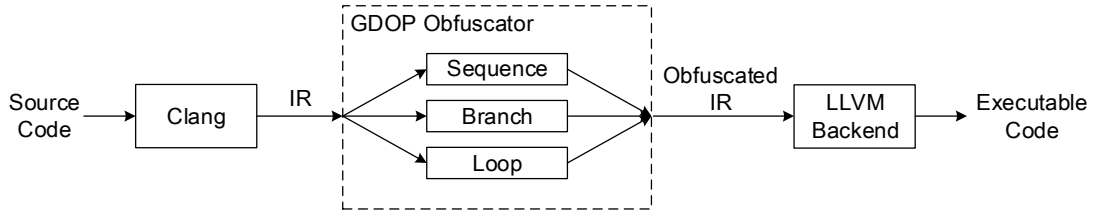


Figure 5.5: The architecture of dynamic opaque predicate obfuscator.

6. Add dynamic opaque predicates as separate basic blocks and according jumps to build correct control flow.

Note that at the third step, we initialize the correlated variables so as to ensure the control flow goes to the left branch at the first iteration. The reason is that we have to make the loop invariant instructions executed at least once at the first iteration of the loop in order to assure all loop invariants loaded, computed and stored correctly. The value of correlated variables may change during the dashed part of the loop body so as to divert the execution flow to each opaque trace. Particularly, when the execution reaching the last iteration of the loop, there is a redundant instruction 2 if the execution follows the right branch. Since instruction 2 is loop independent, it doesn't affect the semantic of the program execution.

5.3 Implementation

Our implementation is based on Obfuscator-LLVM [25], an open source fork of the LLVM compilation suite that aims to improve the software security via code obfuscation and tamper-proofing. The architecture of our system is shown in Fig. 5.5. The generalized dynamic opaque predicate obfuscator (GDOP obfuscator) is surrounded with dashed lines. Basically, our automatic GDOP obfuscator works as a pass in LLVM framework. The workflow contains three steps. First, the LLVM frontend Clang read the source code and translate it into LLVM IR. Second, GDOP obfuscator reads the IR and inserts generalized dynamic opaque predicates to the appropriate location. At last, the LLVM backend outputs the executable program based on the obfuscated IR files.

Particularly, we implement the procedure of inserting generalized dynamic opaque predicates to a straight-line, branch and loop program as three separate

passes, which includes 1251 lines of C++ code in total. We also write a driver program to invoke the three passes so as to insert all kinds of generalized dynamic opaque predicates. In addition, we implement a junk code generator to insert useless code into functions, such as redundancy branches and extra dependencies. Moreover, we provide a compiler option for users to configure the probability for inserting generalized dynamic opaque predicates. For each basic block, our obfuscator generates a random number between zero and one. If the number is smaller than the given probability, it tries to insert generalized dynamic opaque predicates into the basic block; otherwise it skips the basic block.

5.4 Evaluation

We conduct our experiments with several objectives. First, we want to evaluate whether our approach is effective to obfuscate control flow graph. To this end, we measure control flow complexity of GNU Coreutils with three metrics. We also test our tool with a commercial binary diffing tool which is based on control flow graph comparison. Last but not least, we want to prove our approach can defeat the state-of-the-art deobfuscation tool. Our testbed consists of an Intel Core i7-3770 processor (Quad Core with 3.40GHz) and 8GB memory, running Ubuntu Linux 12.04 LTS. We turn off other compiler optimization options by using `-g` option.

5.4.1 Obfuscation Metrics with Coreutils

This section shows our evaluation result of inserting generalized dynamic opaque predicates into the GNU Coreutils 8.23. Since the generalized dynamic opaque predicate is an intra-procedural obfuscation [47], we evaluate it by comparing the control flow complexity of the modified function before and after the generalized dynamic opaque predicate obfuscation. In this experiments, we choose five hot functions in the Coreutils program set by profiling. At the same time, we make sure all the functions containing at least ten basic blocks[1]. After profiling, the five hot functions we select are as follows.

[1]We do not consider dynamic link library functions because our approach takes the target program source code as input.

Table 5.2: Obfuscation metrics and BinDiff scores of hot functions in Coreutils.

Fun	# of Basic Blocks			# of CFG Edges			Cyclomatic Number			Bindiff Score	
	Orig.	50%	100%	Orig.	50%	100%	Orig.	50%	100%	50%	100%
1	43	171	229	62	258	338	21	89	111	0.05	0.02
2	20	75	105	30	114	158	12	41	55	0.02	0.01
3	30	94	120	49	141	177	21	49	59	0.02	0.02
4	46	138	208	80	220	320	36	84	114	0.04	0.01
5	76	272	376	117	425	573	43	155	199	0.05	0.02

1. `get_next`: This function is defined in `tr.c`. It returns the next single character of the expansion of a list.

2. `make_format`: This function is defined in `stat.c`. It removes unportable flags as needed for particular specifiers.

3. `length_of_file_name_and_frills`: This function is defined in `ls.c` for counting the length of file names.

4. `print_file_name_and_frills`: This function is also defined in `ls.c`. It prints the file name with appropriate quoting with file size and some other information as requested by switches.

5. `eval6`: This function is defined in `eval6.c` to handle sub-string, index, quoting and so on.

The metrics that we choose to show the CFG complexity are the number of CFG edges, the number of basic blocks and the cyclomatic number. The cyclomatic number is calculated as $e - n + 2$ where e is the number of CFG edges and n is the number of basic blocks. The cyclomatic number is considered as the amount of decision points in a program [93] and has been used as the metrics for evaluating obfuscation effects [2]. We first insert generalized dynamic opaque predicates into the hot functions with two different probability level: 50% and 100%. After that, we perform functionality testing to make sure our obfuscation is semantics-preserving. Table 5.2 shows the obfuscation metrics of the original clean version and the obfuscated version. The data shows that our dynamic opaque predicate obfuscation can significantly increase the program complexity.

Table 5.3: The result of LOOP detection. Det. refers to the number of detected DOP.

Function	Straight Line DOP			Branch DOP			Loop DOP		
	Total	Det.	Ratio	Total	Det.	Ratio	Total	Det.	Ratio
1	52	3	5.77%	21	0	0.00%	8	0	0.00%
2	28	2	7.14%	15	0	0.00%	6	0	0.00%
3	27	2	7.41%	23	0	0.00%	6	0	0.00%
4	54	5	9.26%	26	0	0.00%	8	0	0.00%
5	82	8	9.76%	52	0	0.00%	14	0	0.00%

To test the control flow graph after our obfuscation is heavily cluttered, we also evaluate our approach with BinDiff[2], which is a commercial binary diffing tool by measuring the similarity of two control flow graphs. We run BinDiff to compare the 50% and 100% obfuscated versions with the original five programs and the similarity score is presented in the fifth column in Table 5.2. The low scores indicate that the obfuscated program is very different from the original version.

5.4.2 Resilience

In this experiment, we evaluate the resilience to deobfuscation by applying LOOP [94], the latest formal program semantics-based opaque predicate detection tool. The authors present a program logic-based and obfuscation resilient approach to the opaque predicate detection in binary code. Their approach represents the characteristics of various opaque predicates with logical formulas and verifies them with a constraint solver. According to the authors, LOOP is able to detect various opaque predicates, including not only simple invariant opaque predicates, but also advanced contextual and dynamic opaque predicates.

In our evaluation, we run two round of 100% obfuscation on the five Coreutils functions and use LOOP to check them. The results are presented in Table 5.3.

As shown in Table 5.3, LOOP can detect very few number of the generalized dynamic opaque predicates inserted in straight-line code but fails to detect all those in branches and loops. We look into every generalized dynamic opaque predicate that is detected by LOOP and find that they are all conventional adjacent dynamic

[2]http://www.zynamics.com/bindiff.html

opaque predicates. We also check verify that LOOP fails to detect the remaining generalized dynamic opaque predicates.

We carefully analyze LOOP's report and find several reasons that lead to LOOP's poor detection ratio on generalized dynamic opaque predicates. First, iterative injection causes LOOP fails to detect majority of the generalized dynamic opaque predicates in straight-line code. Our obfuscation method can be iteratively executed on a candidate function, which means we are able to insert generalized dynamic opaque predicates into the same function several times. Note that each time we choose different correlated variables and different correlated predicates. Therefore, the generalized dynamic opaque predicates that are inserted by the later pass will break the adjacency of those inserted by the previous pass. In addition, junk code injection is another reason that prevents LOOP's detection.

Second, generalized dynamic opaque predicates spread across the branch or loop structure so they naturally break the adjacency property, which causes LOOP detects none of the generalized dynamic opaque predicates in branches and loops. For example, when we execute the loop shown in Fig. 5.3, there are two correlated but not adjacent predicates. They are separated by the instructions in the dashed line and the loop condition. Therefore, the detection method in the LOOP paper fails to detect the generalized dynamic opaque predicates.

5.4.3 Cost

This section presents the cost evaluation of our generalized dynamic opaque predicate obfuscation. We evaluate the cost from two aspects: binary code size and execution time. For binary code size, we measure and compare the number of bytes of the compiled programs that contain the five hot functions. For instance, we compare the size of `tr`'s binary code when inserting generalized dynamic opaque predicates to function `get_next` with different probabilities such as 50% and 100%. For the evaluation of execution time, we record and compare the execution time of clean version and the obfuscated program. We configure the switches and input files so as to ensure the control flow touches the obfuscated function.

Table 5.4 shows the evaluation result. We can observe that our approach slightly increases the binary code size, which is less than 0.7%. Moreover, according to our experiments, the generalized dynamic opaque predicates have a small impact

Table 5.4: Cost evaluation of the dynamic opaque predicate obfuscation.

Fun	Program	Binary Size (Bytes)				Execution Time (ms)		
		Orig.	50%	100%	Ratio	Orig.	50%	100%
1	`tr`	132,084	132,826	133,491	0.53%	2.2	2.2	2.4
2	`stat`	210,864	211,355	211,710	0.20%	4.0	4.0	4.1
3	`ls`	350,076	350,916	351,527	0.21%	23.2	23.4	23.7
4	`ls`	350,076	351,083	351,742	0.24%	23.2	23.3	23.8
5	`expr`	129,696	130,836	131,409	0.66%	0.6	0.6	0.6

Table 5.5: Obfuscation metrics of `sort_files`. R1 and R2 refer to the first and the second round of obfuscation.

Function	# of Basic Blocks			# of CFG Edges			Cyclomatic Number		
	Orig.	R1	R2	Orig.	R1	R2	Orig.	R1	R2
`sort_files`	19	160	405	25	255	539	8	97	136

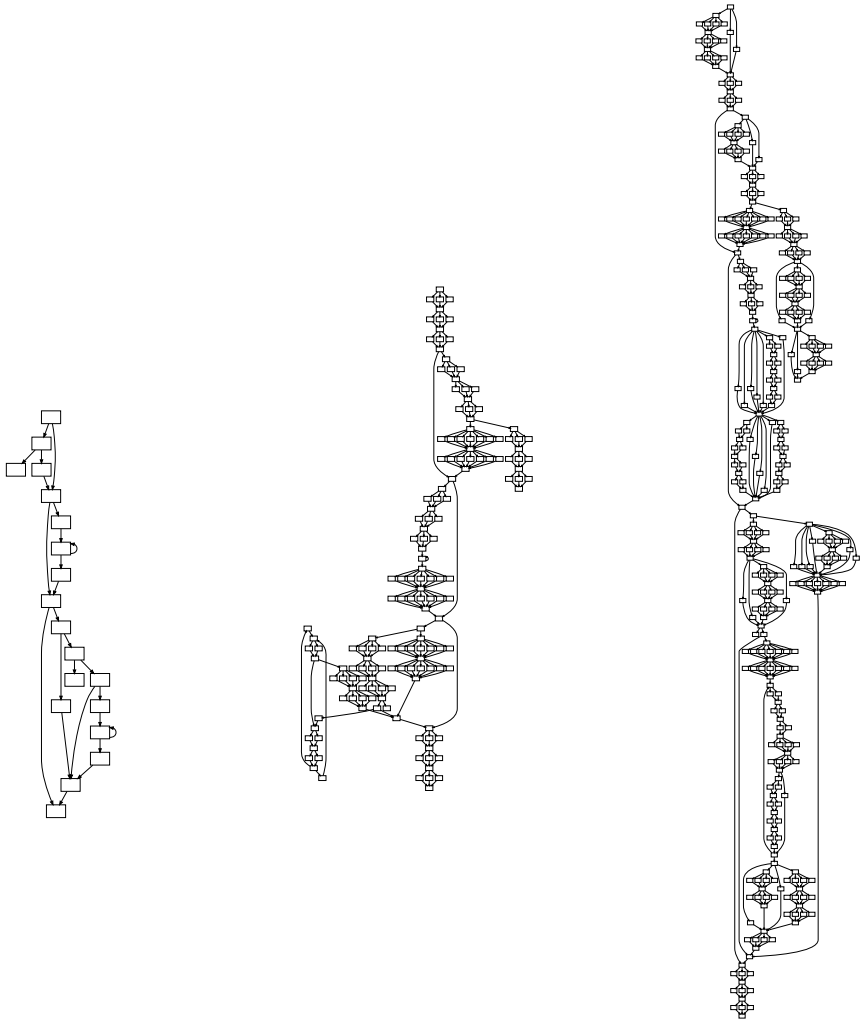
on program performance. The execution time of most programs stays the same when inserting generalized dynamic opaque predicates with 50% probability and increases a little when inserting with 100% probability.

5.4.4 Case Study

As mentioned in Section 5.4.2, our generalized dynamic opaque predicates can be iteratively apply to a candidate program so as to create more obfuscated result. In this section, we provide a case study to show the result of generalized dynamic opaque predicate obfuscation iteration.

The target function is the `sort_files` function in the `ls` program. We choose this function since its CFG size is appropriate and it contains straight-line codes, branches and loops, which are suited for inserting all three categories of generalized dynamic opaque predicates. We perform two rounds of generalized dynamic opaque predicate obfuscation with 100% probability. Table 5.5 presents the same measures as shown in the last section and Fig. 5.6 shows the result CFG. Fig. 5.6(a) shows the original CFG of `sort_files` Fig. 5.6(b) presents the CFG after the first round of dynamic opaque predicate obfuscation. Next, we perform another round of dynamic opaque predicate obfuscation on (b) and the result is shown in Fig. 5.6(c). The comparison of the three CFGs clearly indicates that our generalized dynamic

opaque predicate obfuscation can significantly modify the intra-procedural control flow graph.



(a) The original CFG. (b) The CFG after one round of obfuscation. (c) The CFG after two rounds of obfuscation.

Figure 5.6: Comparison between CFGs after different rounds of dynamic opaque predicate obfuscation.

Chapter 6
Cryptographic Function Detection in Obfuscated Binaries

6.1 Introduction

In this chapter, we present a new approach, *CryptoHunt*, to detect crypto functions inside binary code. This technique helps detect the opaque predicate enhanced by cryptography. Our key idea is to capture the fine-grained semantics of the principal cryptographic transformation iterations along an execution trace. The execution trace is further split into segments according to an enhanced loop abstraction. We then perform bit-precise symbolic execution inside a loop body, and the generated boolean formulas are later used as signatures to efficiently match cryptographic algorithms in obfuscated binaries. Our core technique, *bit-precise symbolic loop mapping*, is effective to revert various data and control obfuscation effects, and also with a much broader detection scope.

In particular, CryptoHunt's detection includes the following main steps. First, we automatically represent the core transformations of a reference cryptographic algorithm (i.e., golden implementation) using boolean formulas. Then, we run the target obfuscated program and record an execution trace. Our enhanced loop abstraction can accurately identify loop structures inside the trace. After that, we run bit-precise symbolic execution to translate the loop bodies into boolean formulas, which are later compared with the reference implementations. However, bit-wise symbolic formula equivalent matching using theorem prover is computationally

The work of this chapter is published in the 38th IEEE Symposium on Security and Privacy [95].

expensive and impractical. To ameliorate this performance bottleneck, we propose a guided fuzzing method to filter out most of the impossible symbolic variable mappings, leaving only about 5% for further verification.

We have evaluated CryptoHunt on a set of synthetic examples collected from GitHub, well-known cryptographic libraries, and malware. We compared CryptoHunt with other six representative tools, and the experiment results are encouraging. In all cases, only CryptoHunt is able to detect commonly used cryptographic functions (e.g., TEA, AES, RC4, MD5, and RSA) under different control and data obfuscation scheme combinations. In addition to obfuscation, skilled malware developers would customize cryptographic algorithms to evade detection [59]. We indeed identified such a non-standard XTEA implementation that reveals a different key schedule constant [60]. Our evaluation shows CryptoHunt is a general and obfuscation-resilient approach, and can be applied to real-word malware analysis and forensics. In summary, we make the following contributions:

- We have proposed a new approach, CryptoHunt, to detect cryptographic functions in obfuscated binaries. Our key solution is to match the principal cryptographic transformation iterations with bit-precise symbolic loop mapping. CryptoHunt exhibits stronger resilience to code obfuscation techniques and a wider detection range.

- We have designed a guided fuzzing method to solve the scalability issue of bit-wise symbolic formula equivalence checking. Our approach greatly reduces the number of possible matches, and can be applied to speed up other semantics-based binary difference analysis methods.

- We have implemented a prototype of CryptoHunt. The source code is publicly available at `https://github.com/s3team/CryptoHunt`.

The rest of the chapter is organized as follows. Section 6.2 presents an overview of CryptoHunt. Section 6.3 to 6.8 discuss the details of each step in our method. Section 6.9 describes our implementation details. We present our evaluation results in Section 6.10.

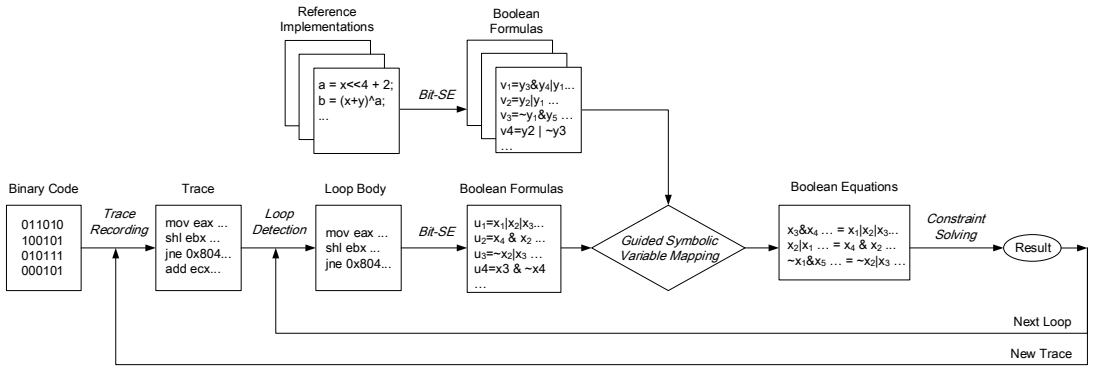


Figure 6.1: An overview of CryptoHunt's workflow. The words in italics represents CryptoHunt's key components, and "*Bit-SE*" stands for bit-precise symbolic execution.

6.2 Overview

The shortcomings of existing work inspire us to design a new general solution to detect cryptographic algorithms and variations in obfuscated binaries. Instead of searching syntactical signatures, we attempt to capture the fine-grained semantics of the principal cryptographic transformations. Figure 6.1 illustrates CryptoHunt's workflow, which contains the following key steps.

1. *Execution trace generation.* Since dynamic analysis has previously been demonstrated to be effective in control flow deobfuscation [34, 80] and analyzing self-modifying code [96], our study continues dynamic detection direction. We first run the target binary code and record the execution trace, which contains detailed runtime information.

2. *Loop body identification.* Like many dynamic detection methods [55, 57], we identify loop structures to narrow down search scope. The reason is cryptographic algorithms consist of a large of repeated transformations, which are typically implemented as loops.

3. *Bit-precise symbolic execution.* Attackers can impede further analysis by transforming (I/O) parameters with data obfuscation schemes. To revert data obfuscation effects, our key idea is to represent loop I/O relations with bit-precise symbolic execution. In this way, loop input parameters are

```
1  void encrypt (uint32_t* v, uint32_t* k) {
2
3    /* v: plain text, k: key */
4    uint32_t v0 = v[0], v1 = v[1], sum = 0, i;
5    uint32_t k0 = k[0], k1 = k[1], k2 = k[2], k3 = k[3];
6
7    /* delta: a key schedule constant */
8    uint32_t delta = 0x9e3779b9;
9
10   for (i = 0; i < 32; i++) {        /* main loop */
11     sum += delta;
12     v0  += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
13     v1  += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
14   }
15
16   v[0] = v0; v[1] = v1;             /* cipher text */
17 }
```

Figure 6.2: A reference implementation of TEA.

expressed as boolean variables, which is the only atomic data type. The output parameters are represented as a set of boolean formulas.

4. *Variable mapping and comparison.* We propose a guided fuzzing approach to efficiently find whether a symbolic formula is equivalent to a reference implementation. Only a small portion of symbolic formulas need to be further verified by a theorem prover.

We will present the details of each step in the following sections.

6.3 Reference Formula Generation

We compare boolean formulas from the target execution trace with those from the reference implementation. In this section, we describe how to generate boolean formulas from the reference implementation. We choose standard cryptographic algorithm implementations (e.g, widely-used OpenSSL crypto library) as the reference. Since we have access to the C source code of the reference implementation, we first iterate C code structures to identify the principal cryptographic transfor-

mation iterations with CIL [97]. The main loop in Figure 6.2 shows such a key transformation iterations in TEA cipher. Section 6.10.1.4 will provide more details about the key transformation that we capture in commonly used cryptographic algorithms.

Next, we compile the source code into an executable and run it to record a trace. Then we perform bit-precise symbolic execution for the loop body and generate a set of boolean formulas, which will be used as semantic detection signatures later. More details about trace recording and bit-precise symbolic execution will be presented in Section 6.4 and 6.6. The reference formulas usually have two attributes. One is that they are compact to describe the most representative feature of a given cryptographic algorithm. It is not necessary to depict the whole transformation of the algorithm in the reference formula. The other attribute is abstraction, which means the formulas are independent of a specific implementation. Security analysts can generate the reference formulas by just reading the algorithm description. The feature described by the formula should be encoded into all implementations of the algorithms. Taking TEA as an example, the reference formulas are as follows:

$$y = ((x_1 << 4) + x_2) \oplus (x_1 + x_3) \oplus ((x_1 >> 5) + x_4) + x_5$$

Here we group a set of bit symbols as $x_1, ..., x_5$ for the easy presentation purpose. Note that the concrete variable *sum* in Figure 6.2 is represented as a symbolic variable x_3. It is because the value of *sum* derives from a key schedule constant, *delta*. Skilled malware authors can customize this constant value to produce implementation variations, which will bypass our detection. To make the reference formula more flexible, we substitute *sum* as a symbol as well. We will discuss such a non-standard XTEA implementation we identify in Section 6.10.2.

6.4 Execution Trace Recording

When analyzing a target binary program, we first record its execution trace. CryptoHunt's trace record component is built based on Pin, a dynamic binary instrumentation framework developed by Intel [85]. All instructions except system call are recorded during the run time. The trace includes the following information.

1. The memory address of each instruction

2. The machine instruction name (opcode) which describes its operation, such as `load` or `mov`

3. The source and destination operands of the instruction, which could be an immediate value, a register name, or a memory address

Malware authors commonly apply various binary packing tools to hide the real code and then recover the real malicious code during execution. Recording binary unpacking routine will bring many useless instructions. Our purpose is to detect the cryptographic algorithm inside obfuscated binaries. To this end, we utilize generic runtime unpacking techniques [96, 98] to renew trace recording when the execution flow returns to the original entry point.

6.5 Loop Body Identification

From the previous step, we obtain an execution trace of the target program. As mentioned before, CryptoHunt detects cryptographic code inside loop structures. In this section, we present how to identify loop bodies inside a trace. Our method extends Calvet's loop detection algorithm [55] so as to detect more categories of loops.

First, we clarify the loop definition in this paper. A loop is a sequence of instructions that meets one of the following requirements.

1. The opcode of the sequence of instructions repeat at least one time.

2. The instruction sequence ends with a conditional or unconditional jump instruction jumping to the beginning of the instruction sequence.

Figure 6.3 shows two trace examples according to the loop definition. In Figure 6.3(a), the instruction sequence `[1,2,3,4]` repeat at least two times, which meets the first loop definition. This loop form is usually corresponding to an unrolled loop by compiler optimization. In Figure 6.3(b), the trace contains a conditional jump instruction `jne 8048100`, which jumps to an instruction that has been executed. So it meets the second rule in our loop definition. Notice that although the control flow jumps back to a previously executed instruction, the following instruction sequence is not as same as the previous one. This is because

there might be conditional branches inside a loop body, which leads to execution of different instructions in each loop iteration. In practice, many loops in an execution trace fall into the second category. One example in cryptographic algorithm is the modular exponentiation implementation in RSA. It is typically implemented as a loop containing two branches. One branch is a multiplication and the other one is a squaring and a multiplication. In every iteration, the execution flow takes one branch based on the bit of the exponent being referenced so the iterations of the same loop could be different. Calvet's loop detection algorithm [55] only detects the case in Figure 6.3(a). Our loop identification method covers both cases in Figure 6.3.

We provide a brief description of the loop identification algorithm. First, when scanning the first category of loops in a trace, we reuse the loop detection algorithm in Calvet's work [55]. One extension in our loop identification algorithm is matching function call/return instructions pairs. Function calls could break the loop definition in Figure 6.3(a). For example, calling the same function with different parameters could result in different control flow in the function. Therefore, we need to eliminate the function calls' interference inside the execution trace. We try to match the function call and return instruction during the loop identification. The matching procedure is one scanning pass on the execution trace. We maintain a stack to simulate nested function calls inside the trace. During the matching process, when a call instruction is seen in the scanning procedure, we push it to the stack and record the entrance address. When a return instruction is seen, we pop the call instruction from the stack and replace the whole function body with the call instruction and its entrance address. In this way, during the loop identification, function calls with the same entrance address are recognized as the same instructions, which prevent the function calls' interference in the loop identification algorithm.

In order to identify the second category of loops, we seek for the jump instructions whose destination instruction has been executed in the trace. When such a jump instruction is identified, we mark the address range between the jump instruction and its destination. If the instruction following the jump is the destination instruction, we identify the range as a loop. The process is repeated until the next instruction of the jump is not its destination. Note that we could identify different iterations of the same loop in this category. For example, in Figure 6.3(b), `[1,2,3,4]` and `[1,5,6]` are two iterations of the same loop. By computing the hash value of each

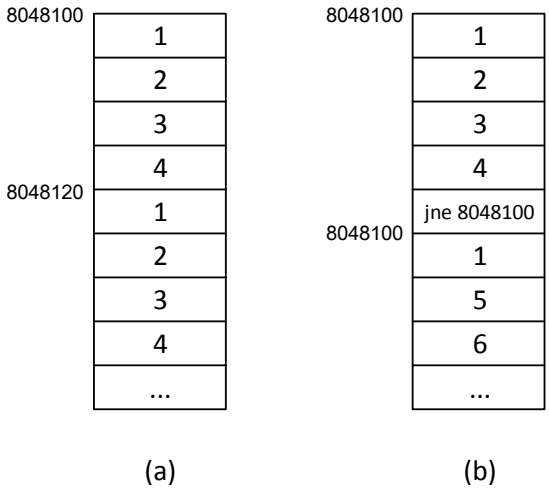


Figure 6.3: Loop identification in an execution trace.

loop iteration, we can distinguish and only record the different iterations for future analysis. For the sake of efficiency, the identification of the second loop category is processed together with the first category.

Moreover, we also identify nested loops by folding all detected loop body iterations. Figure 6.4 shows the folding procedure. In Figure 6.4(a), we identify the repeated instruction sequence `[2,3]` as the innermost loop L_1. Then we fold all iterations of L_1 and replace them with a pseudo instruction named `L1` and continue the loop identification as shown in Figure 6.4(b). In the folded trace, we identify the repeated instruction sequence `[1,L1,4]` as the outer loop L_2. Similarly, all iterations of L_2 are folded and replaced by the pseudo instruction `L2`. The final folded trace is shown in Figure 6.4(c).

After all, the output of loop identification is a set of different loop iterations. Since the number of candidates could be very large, we apply some crypto algorithm specific heuristic methods to filter out non-related loop iterations. Since cryptographic algorithms usually contain intensive bitwise operations, one heuristic method is counting the number of bitwise instructions inside a loop [99]. Another heuristic method is using the absolute entropy of the memory regions accessed in the loop body. It is because that encrypted data is considered to have a high information entropy [51]. The loop iterations after filtering are passed to the following phases for future analysis.

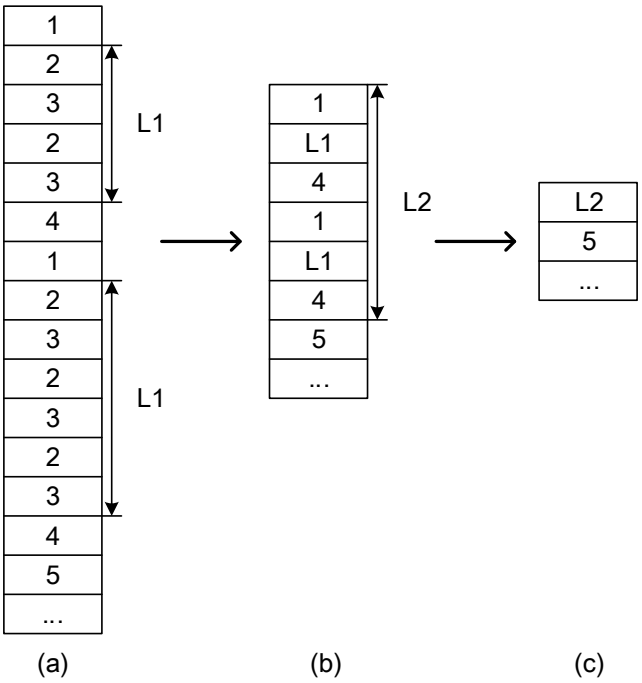


Figure 6.4: Nested loops identification.

6.6 Bit-precise Symbolic Execution in Loop

After identifying loops in the execution trace, we extract each loop body and perform bit-precise symbolic execution to transform them into boolean formulas. In our method, we analyze the loop body, which is only one iteration of the loop. We first identify the free output variables in the loop body and then perform backward slicing from each output variable. In each slice, we can backtrack to the input variables of the loop body. We claim the input variables which meet the following conditions as free input variables and mark them as symbols.

1. The variable is loaded from memory.

2. The variable is not a loop invariant. Since the execution trace includes all run-time information, we can check whether a variable is a loop invariant by comparing different loop iterations.

After that, we symbolically run each slice so as to transform them into a boolean formula, which is composed of a series of boolean functions. Each function is a transformation which takes multiple input variables and generates one output.

Particularly, we transform each free variable into boolean variables. For example, if a free variable is in a 32-bit register, it is transformed to 32 boolean individual variables. Therefore, with bit-precise symbolic execution, we transform the operations associated with the output variables into a boolean formula, which accurately describes the semantics of the instructions inside a loop body.

One benefit of bit-precise symbolic execution is it reveals the fine-grained semantic meaning of the operations inside a loop body so as to resist obfuscation techniques. This feature makes our method outperforms lots of previous work. For example, the current research work Aligot [55] utilize the input/output relation to identify cryptographic functions. One limitation of this category of research is that the parameters in the target program must be exactly same as the parameters in the reference implementation. It is because that they treat the whole loop body as a "black box" without looking into the details inside. In practice, simple data obfuscation such as data aggregation and data split can easily work around Aligot. We present an example in Figure 6.5.

Figure 6.5(a) shows the normal program before the data obfuscation. Variables `a` and `b` are two input variables for the `while` loop. If we already know `a` and `b` are small integers, which will not use the higher bits of the 32 bits, we can aggregate the two variables into 32 bits variable `X` as shown in Figure 6.5(b). Therefore, the `while` loop only has one input variable `X`. Notice that `X` is not equivalent to either `a` or `b`. As a result, cryptographic detection tools based on input and output relation such as Aligot cannot identify those two programs are semantically equivalent. Similarly, we can also split the variable `a` into two variables `a1` and `a2` as shown in Figure 6.5(c) and the input and output data of the `while` loop is also obfuscated. What's more, there are plenty of encoding obfuscation in this category, such as the obfuscation using homomorphic functions [100] and variable merging [43].

On the other hand, bit-precise symbolic execution provides a perfect and final solution for this problem. By translating the operations into boolean formulas, we can compare the fine-grained semantics of different loop bodies. For instance, if we translate the `while` loops in Figure 6.5(a) and (b) into boolean formulas, we will find that the two sets of formulas are essentially doing the same task.

```
int a = f();
int b = g();
...
while (...) {
  m = a << 4;
  n = b * 5;
...
}
```

(a) Normal program.

```
struct {
  int a : 15;
  int b : 17;
} X;

/* aggregate a,b to X */
X.a = f();
X.b = g();
...
while (...) {
  m = X.a << 4;
  n = X.b * 5;
...
}
```

(b) Data aggregation.

```
a = f();
b = g();

/* split a to a1, a2 */
short a1 = a & 000fffff;
short a2 = a >> 20 & 00000fff;
...
while (...) {
  int aa = (int) a2 << 20 | a1;
  m = aa << 4;
  n = b * 5;
...
}
```

(c) Data split.

Figure 6.5: An example of data obfuscation.

6.7 Guided Symbolic Variable Mapping

The bit-precise symbolic execution in the last section output a group of boolean formulas. In this section, we compare these formulas with the reference formulas so as to decide whether they are equivalent. Since each input and output variable is transformed into boolean variables, typically there are dozens of input and output

variables. When comparing those formulas, the key problem is mapping the input variables in target formulas to those in the reference formulas. In previous related work, the mapping is mainly done by permutation and then using a theorem prover to check them one by one.

However, the number of variables in our work is significantly larger so simple permutation will cause serious performance issue. Therefore, we propose a new method to quickly find the possible variable mappings and filter out the impossible ones. In another word, the mapping procedure itself can partially verify the formula's semantics before applying the theorem prover.

6.7.1 Motivation

Before describing the detail matching algorithm, first we provide an example to show why we need a mapping algorithm. Suppose we are comparing two loop bodies. One is from the target program and the other one comes from the reference program. The operations in both loop bodies have been translated to two sets of boolean functions as shown in function set 6.1 and 6.2. Here we call a set of boolean functions as a *formula*. In this example, we suppose that the target and reference program both include three input boolean variables and two output variables. Particularly, formula F is extracted from loop bodies in the target execution trace and G is from the reference program. In formula F, x_1, x_2, and x_3 are input variables, u_1 and u_2 are output variables, and f_1 and f_2 are the boolean functions that compute the output variable value based on the inputs. Similarly, formula G shows input/output variables and functions in the reference program. Notice that here we use x and y to distinguish the input variables in the target program and the reference program.

$$F = \begin{cases} u_1 = f_1(x_1, x_2, x_3) = x_1 \wedge x_2 \vee x_3 \\ u_2 = f_2(x_1, x_2, x_3) = \neg x_1 \vee \neg x_3 \wedge x_2 \end{cases} \tag{6.1}$$

$$G = \begin{cases} v_1 = g_1(y_1, y_2, y_3) = \neg(y_1 \wedge y_2) \wedge y_3 \\ v_2 = g_2(y_1, y_2, y_3) = y_1 \vee (y_2 \wedge y_3) \end{cases} \tag{6.2}$$

In order to check whether the two formulas are semantically equivalent, we need to find out which input variable in formula F is identical to the input variable in

G, and also the output variables. In another word, we need to find two variable mappings as shown in Figure 6.6.

(a) Input mapping. (b) Output mapping.

Figure 6.6: Variable mapping.

Assuming the mappings in Figure 6.6 have been found, we can build a boolean equation set 6.3 to check whether F and G are equivalent. Multiple methods such as fuzz testing and theorem proving can be applied to verify the equation set. If every equation in the set always holds, it proves that formula F and G are equivalent, which means the loop body in the target program is equivalent to the reference. As a result, the target program includes a cryptographic function implementation. We check all loop bodies and report finding a cryptographic algorithm when one loop body matches.

$$\begin{cases} x_1 \wedge x_2 \vee x_3 = \neg(x_2 \wedge x_3) \wedge x_1 \\ \neg x_1 \vee \neg x_3 \wedge x_2 = x_2 \vee (x_3 \wedge x_1) \end{cases} \tag{6.3}$$

6.7.2 Definitions

Starting from this section, we present the formal mapping algorithm. First, we introduce the formal definitions of the concepts used in our algorithm.

Definition 1 *We define a boolean function $f(x_1, x_2, \ldots, x_n)$ as a mathematical function that takes n boolean arguments (inputs) and returns one boolean result (output).*

Definition 2 *We define a boolean formula $F_{n,m}$ which has n inputs and m outputs as a function set that includes m boolean functions, each of which has n inputs:*

$$F_{n,m} = \begin{cases} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \cdots \\ f_m(x_1, x_2, \ldots, x_n) \end{cases}$$

Definition 3 *Given a boolean function $f(x_1, x_2, \ldots, x_n)$, we define its* Input Identity Matrix *as a $n \times n$ matrix:*

$$I_{n,n} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} = \begin{pmatrix} \vec{x}_1^T \\ \vec{x}_2^T \\ \vdots \\ \vec{x}_n^T \end{pmatrix}$$

where $\vec{x}_i^T$ is the ith row vector.

In an input identity matrix, each row vector $\vec{x}_i^T$ represents one input by setting only one variable to 1 and the rest to 0. An input identity matrix enumerates all possibilities of these inputs.

Definition 4 *Given a boolean formula $F_{n,m}$, we define its* Output Matrix *as an $n \times m$ matrix:*

$$M_{n,m}^O = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}$$

where

$$a_{ij} = f_j(\vec{x}_i^T),\ i = 1, 2, \ldots, n \text{ and } j = 1, 2, \ldots, m.$$

In an output matrix, each row is the outputs by feeding the corresponding row vector in the input identity matrix into every boolean function. The insight is that, each row of the output matrix corresponds to one input variable and each column corresponds to one output variable. A permutation of rows in a output matrix is a permutation of the input variable. Similarly, a permutation of columns in a output matrix is a permutation of output variables. Therefore, the mapping problem is essentially equivalent to the following problem:

Can we transform one output matrix to the other by permuting rows and columns?

This is the key idea in our mapping algorithm. Notice that permuting rows and columns still keep the sum of each row or column unchanged. This feature provides a hint for mapping the rows and columns, which correspond to the input and output variables. So we go ahead to define the *row sum vector* and *column sum vector* in an output matrix.

Definition 5 *The* row sum vector $\vec{rv}$ *and* column sum vector $\vec{cv}$ *of an output matrix* $M^O_{n,m}$ *are defined as follows.*

$$\vec{rv} = \begin{pmatrix} \sum_{j=1}^{m} a_{1j} \\ \sum_{j=1}^{m} a_{2j} \\ \vdots \\ \sum_{j=1}^{m} a_{nj} \end{pmatrix}, \vec{cv} = \begin{pmatrix} \sum_{i=1}^{n} a_{i1} \\ \sum_{i=1}^{n} a_{i2} \\ \vdots \\ \sum_{i=1}^{n} a_{im} \end{pmatrix}$$

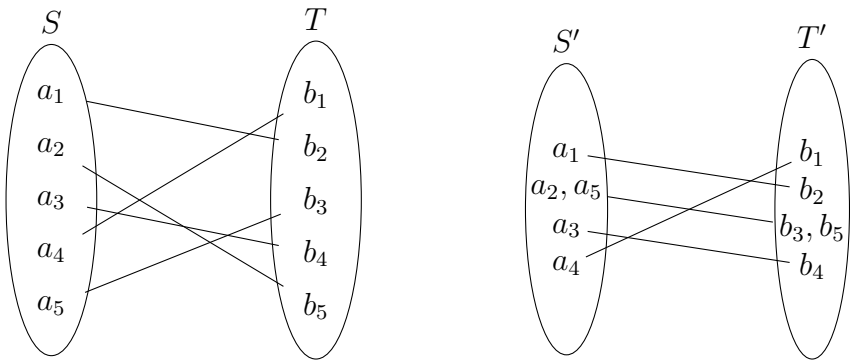
Each element of the row sum vector is the sum of the corresponding row in $M^O_{n,m}$. Essentially it describes the fact that how many outputs is evaluated to 1 when setting a specific input to 1 and leave the rest to 0. Similarly, a column sum vector describes how many output variables are set to 1 in each column of $M^O_{n,m}$. $\vec{rv}$ and $\vec{cv}$ are used for computing the mapping in a given output matrix.

For example, given an output matrix $M^O_{2,3}$, its $\vec{rv}$ and $\vec{cv}$ are shown as follows.

$$M^O_{2,3} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}, \vec{rv} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \vec{cv} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}$$

So far we have defined the concepts related to inputs and outputs of formulas. Since our objective is to find the mapping between two formulas' inputs and outputs, we need to clarify the concept of mapping in this paper. There are two types of mapping, *full mapping* and *partial mapping*. As shown in Figure 6.7(a), a *full mapping* means every element in one set has been mapped to a unique element in the other set. A *partial mapping* means we only find mappings for partial elements in one set. Taking Figure 6.7(b) as an example, we have found mappings for the

elements a_1, a_3 and a_4 in S'. However, the mapping for a_2 and a_5 is still not decided. Possible mappings are $a_2 \mapsto b_3, a_5 \mapsto b_5$ or $a_2 \mapsto b_5, a_5 \mapsto b_3$.



(a) A full mapping. (b) A partial mapping.

Figure 6.7: Mapping examples.

6.7.3 Algorithm Description

We have introduced the basic concepts that are needed to formalize the algorithm. As mentioned above, we transform the variable mapping problem to the output matrix mapping problem; that is, given an input identity matrix, a variable mapping exists if and only if one output matrix can be transformed to the other by permuting rows and columns.

Based on this idea, we propose the variable mapping algorithm, which is described in Algorithm 2. Briefly speaking, the algorithm recursively seeks for all possible mappings based on a series of specific inputs. The method is *complete*, which means if a mapping exists, it must appear in the result. It is possible that the mapping algorithm generates some false positive mappings and they will be checked by the following verification steps.

Given two boolean formulas F_1 and F_2, the high-level panorama of the mapping algorithm is shown as follows. We provide an example to show the mapping algorithm step by step in the following section.

1. Feed the Input Identity Matrix I into F_1 and F_2 respectively and generate two output matrix M_1^O and M_2^O.

2. Create the row and column sum vectors for M_1^O and M_2^O and check them using heuristic constraints.

3. Create mappings based on the row and column mappings. Check whether the mappings are consistent.

4. If one variable is mapped, add it to the mapped list. Otherwise permute building mappings for the elements.

5. Randomly create new inputs based on the mapped variables.

6. Recursively call VarMapping to map the remaining inputs and outputs.

6.7.4 Example

Last section describes an overview of the mapping algorithm. Now we provide an example to show the whole procedure in details. We still use F and G as shown in formula 6.1 and 6.2. We initiate the partial mapping list L set as follows. M_{in} and M_{out} are initiated as empty.

$$\{x_1, x_2, x_3\} \mapsto \{y_1, y_2, y_3\}$$
$$\{u_1, u_2\} \mapsto \{v_1, v_2\}$$

First, since M_{in} is empty, there is no mapped variable. We generate the input identity matrix for all input variables. After that we create the output matrix accordingly. For the ease of understanding, we show the procedure in equation 6.4 and the matrix in 6.5.

$$\begin{cases} \{x_1 = 1, x_2 = 0, x_3 = 0\} \Rightarrow \{u_1 = 0, u_2 = 0\} \\ \{x_1 = 0, x_2 = 1, x_3 = 0\} \Rightarrow \{u_1 = 0, u_2 = 1\} \\ \{x_1 = 0, x_2 = 0, x_3 = 1\} \Rightarrow \{u_1 = 1, u_2 = 0\} \end{cases} \tag{6.4}$$

$$M_1^I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, M_1^O = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{6.5}$$

Following the same method, we feed the inputs into G and the result is shown in 6.6 and 6.7.

$$\begin{cases} \{y_1 = 1, y_2 = 0, y_3 = 0\} \Rightarrow \{v_1 = 1, v_2 = 0\} \\ \{y_1 = 0, y_2 = 1, y_3 = 0\} \Rightarrow \{v_1 = 0, v_2 = 0\} \\ \{y_1 = 0, y_2 = 0, y_3 = 1\} \Rightarrow \{v_1 = 0, v_2 = 1\} \end{cases} \tag{6.6}$$

$$M_2^I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, M_2^O = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \tag{6.7}$$

Based on M_1^O and M_2^O, we create the row and column sum vectors as follows.

$$r\vec{v}_1 = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, c\vec{v}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, r\vec{v}_2 = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, c\vec{v}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

The sorting result of $r\vec{v}_1$ and $r\vec{v}_2$ shows they are equivalent and so do $c\vec{v}_1$ and $c\vec{v}_2$. Therefore, we move on to the next step to connect the rows that have the same number in row sum vectors. So $x_1 \mapsto y_2$ and $\{x_2, x_3\} \mapsto \{y_1, y_3\}$ is created and added to L. Similarly, we create connections between columns. As a result, L is updated as follows.

$$\begin{gathered} \{x_1, x_2, x_3\} \mapsto \{y_1, y_2, y_3\} \\ x_1 \mapsto y_2 \\ \{x_2, x_3\} \mapsto \{y_1, y_3\} \\ \{u_1, u_2\} \mapsto \{v_1, v_2\} \end{gathered}$$

We reduce the connections in L by intersection operations. L can be normalized to the following form.

$$\begin{gathered} x_1 \mapsto y_2 \\ \{x_2, x_3\} \mapsto \{y_1, y_3\} \\ \{u_1, u_2\} \mapsto \{v_1, v_2\} \end{gathered}$$

After that, in Line 27 of the mapping algorithm, we find a mapping $x_1 \mapsto y_2$. So we remove it form L and add it to M_{in} since it is a connection of the input variables. Then we recursively call VarMapping again using the updated L and M_{in}.

In the second call of VarMapping, we generate random input for variables in M_{in}. Notice that the connected variables must have the same value. For example, we generate $x_1 = 1, y_2 = 1$. We still generate input identity matrix for the remaining input variables in L. Therefore, the input and output matrix are as follows.

$$M_1^I = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}, M_1^O = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

$$M_2^I = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}, M_2^O = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

Similarly, we create the row and column sum vectors for M_1^O and M_2^O.

$$\vec{rv}_1 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \vec{cv}_1 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \vec{rv}_2 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \vec{cv}_2 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

The vectors pass the sorting check as before. By updating and reducing L, the result is as follows.

$$x_2 \mapsto y_3$$
$$x_3 \mapsto y_1$$
$$u_1 \mapsto v_2$$
$$u_2 \mapsto v_1$$

After moving the mapping in L to M_{in} and M_{out}, L is empty. So the final result will be returned in M_{in} and M_{out} when calling VarMapping next time. The final mapping result is shown in Figure 6.8.

(a) Input mapping. (b) Output mapping.

Figure 6.8: Final result of variable mapping.

Based on the variable mapping information, we can produce the equation set for the following verification procedure.

$$\begin{cases} x_1 \wedge x_2 \vee x_3 = x_3 \vee (x_1 \wedge x_2) \\ \neg x_1 \vee \neg x_3 \wedge x_2 = \neg(x_3 \wedge x_1) \wedge x_2 \end{cases} \tag{6.8}$$

In this example, our mapping algorithm generates one candidate variable mapping. Being compared with the permutation, which will generate $3! \times 2! = 12$ inputs

and outputs combinations, our method reduces the number of candidates. Notice that in our example we only show three input variables and two output variables. Since permutation is a factorial function, when the number of variables increases, the number of permutation will grow very fast. Our mapping algorithm can filter out unmapped formulas and significantly reduce the number of candidates.

6.7.5 Algorithm Analysis

In Section 6.7.2, we state that the variable mapping problem is equivalent to the following question:

Can we transform one output matrix to the other by permuting rows and columns?

In fact, the matrix permutation problem can be further proved to be equivalent to bipartite graph isomorphism problem. The proof is shown as follows.

Given two output matrices M_1^O and M_2^O, which both have n rows and m columns. Supposing they are adjacent matrices, we construct two bipartite graphs G_1 and G_2. G_1 has n blue vertices $b_1, b_2, \ldots, b_n$ and m red vertices $r_1, r_2, \ldots, r_n$. In M_1^O, if the element at the ith row and jth column is 1, an edge is created connecting the vertex b_i and r_j in G_1. Similarly, G_2 is constructed from M_2^O. Permutation of the rows and columns in M_1^O and M_2^O only changes the order of vertices. It does not change the topological structure of the corresponding graph. Therefore, the problem of checking whether M_1^O can be transformed to M_2^O by permuting rows and columns is equivalent to checking whether G_1 and G_2 are isomorphic.

The bipartite graph isomorphism problem is not known to be solvable in polynomial time, so the matrix permutation problem in this paper has the same complexity. Therefore, in general our guided-fuzzing method does not always guarantee to return the mapping result in a reasonable time. In some extreme cases, our method could lead to time out. For example, when there are many redundant variables in both of the two matrices, our method has to permute all possible mappings between each pair of them, because flipping these redundant variables does not change the out matrix. The permutation could cause time out. However, in practice the reference formula in our method is from an open source crypto library, we can guarantee it does not contain those redundant variables. If

only the target program has redundant variables, our method can still return the correct mapping result.

6.8 Verification

In this section, we present the method to verify the boolean equations generated by the previous step. Basically, we use two methods, fuzz testing and theorem proving. Fuzz testing is quick but the result is not sound; that is, passing fuzz testing does not mean the equations always hold. Theorem proving is slow but the result is sound. Therefore, we use fuzz testing as the first round to filter out some candidates and apply theorem proving to the remains.

We randomly generate some inputs and feed them into the boolean equation set to test whether they hold. In our practical experience, this is an easy and quick way to get rid of many wrong mappings and give a partial equivalence checking. For example, if all mapping candidates do not pass fuzz testing, we can safely decide the two sets of formulas are semantically different.

After the equation sets pass fuzz testing, we utilize a theorem prover to prove the formulas. If the formulas hold, we claim that the target program and the reference program are semantically equivalent; otherwise they are different.

6.9 Implementation

We build a tool named *CryptoHunt* as an implementation of the idea in this paper. The trace logging component is built based on Intel's Pin DBI framework [85] (version 2.12) with 945 lines of code in C/C++. The loop identification component is implemented with 374 lines of Perl code. CryptoHunt's bit-precise symbolic execution is built based on BAP [37] (version 0.8), which is used to lift x86 instructions to the BAP IL and further into boolean formulas in CVC format. We also built a framework to implement the formula mapping algorithm, fuzz testing, and other formula analysis, which includes 1700 lines of C/C++ code. Moreover, we adopt STP [83] as the theorem prover. For the convenience of future research, we have released CryptoHunt source code at `https://github.com/s3team/CryptoHunt`.

Table 6.1: Cryptographic algorithm categories.

Category	Algorithm
Block cipher	TEA and AES
Stream cipher	RC4
Hashing algorithm	MD5
Asymmetric cipher	RSA

6.10 Evaluation

In this section, we evaluate CryptoHunt from two main aspects: *effectiveness* and *performance*. Particularly, we conduct our experiments to answer the following research questions (RQs).

1. **RQ1:** Is CryptoHunt effective to detect widely used cryptographic algorithms in obfuscated binaries? (*effectiveness*)
2. **RQ2:** How many false positives can CryptoHunt produce? (*effectiveness*)
3. **RQ3:** How much overhead can CryptoHunt's dynamic detection approach introduce? (*performance*)

As the answer to RQ1, we compare CryptoHunt with other peer tools using crypto projects collected from GitHub with different obfuscation techniques. We also evaluate them on malware samples. In RQ2, we use normal programs such as core utilities, compression tools, and server programs to test the false positives. In response to RQ3, we report CryptoHunt's performance data such as running time, number of identified loops, and number of STP queries. We also report the performance improvement introduced by our guided fuzzing approach.

6.10.1 Answer to RQ1: Crypto Libraries

6.10.1.1 Dataset

We first test CryptoHunt with commonly used cryptographic algorithms from four categories (see Table 6.1). We choose TEA and AES as block cipher examples. Tiny Encryption Algorithm (TEA) [101] is a simple block cipher, which is frequently

adopted by malware authors to hide malicious intent; while the Advanced Encryption Standard (AES) [102] is a more complicated block cipher, which has been used by crypto-ransomware to encrypt victim's documents. RC4 is chosen as the stream cipher candidate. It is used by standards such as IEEE 802.11 within WEP (Wireless Encryption Protocol) using 40 and 128-bit keys. We choose MD5 [103] as the hashing algorithm since it is widely used on the Internet for software integrity checking. At last, we use RSA [104] as the asymmetric cipher candidate. RSA is one of the first practical asymmetric ciphers in the world and is widely used for secure data transmissions. In practice, programmers usually take advantage of existing cryptographic libraries when they need encryption/decryption function. This is due to two reasons. First, cryptographic algorithms are highly standardized. Many cryptographic libraries such as OpenSSL and Libgcrypt already have correct implementations, so there is no need for normal programmers to re-implement them. The other reason is that cryptographic algorithms are complicated and difficult to implement. It is common that user-implemented cryptographic algorithms are buggy. Therefore, as one common scenario of using cryptographic algorithms, we evaluate CryptoHunt on popular cryptographic libraries. In our evaluation, we test two open source libraries: OpenSSL[1] and Libgcrypt[2]. OpenSSL and Libgcrypt are both widely used in real world software systems such as web server, email client and web browser. Our purpose is to detect commonly used cryptographic algorithms provided by standard libraries. To this end, we collect 25 open source projects from GitHub[3]. For each crypto algorithm in Table 6.1, we collect 5 projects. All the 25 projects reuse cryptographic functions from either OpenSSL or Libgcrypt. The configuration of our testbed machine is shown as follows.

- CPU: Intel Core i7-3770 processor (Quad Core with 3.40GHz)
- Memory: 8GB
- OS: Ubuntu Linux 14.04 LTS
- Compiler: GCC 4.8.4
- Crypto Libraries: OpenSSL 1.1.0-pre3, Libgcrypt 1.6.4

[1]https://www.openssl.org/
[2]https://www.gnu.org/software/libgcrypt/
[3]https://github.com

6.10.1.2 Peer Tools

We compare CryptoHunt with six cryptographic code detection tools: CryptoSearcher, Findcrypto2, Signsrch, DFGIsom, Kerchkhoffs, and Aligot. These six tools represent both static and dynamic detection directions. CryptoSearcher [105] is an assembly tool that identifies cryptographic programs by static signatures. Similarly, both Findcrypto2 [106] and Signsrch [107] are IDA [108] plug-in tools and search static signatures for cryptographic function detection. DFGIsom [54] statically identify symmetric cryptographic algorithms and their parameters inside binary code based on Data Flow Graph (DFG) isomorphism[4]. Kerchkhoffs [57] is a trace analysis tool, which provides methods to reconstruct high-level information from a trace, for example control flow graphs or loops, to detect cryptographic algorithms and their parameters. The advanced detection tool, Aligot [55], relies on identifying unique input-output relations at loop boundary.

6.10.1.3 Obfuscation Options

To obfuscate cryptographic algorithm implementations, we rely on a state-of-the-art compile-time obfuscation tool, Obfuscator-LLVM [110], which supports popular obfuscation transformations [1,111]. We have extended Obfuscator-LLVM to include three obfuscation options, *N, O1,* and *O2*, which specify different obfuscation levels. The details of the obfuscations included in each option are listed as follows.

1. *N*: The obfuscator does not perform any obfuscation.

2. *O1*: The obfuscator performs simple instruction-level obfuscation and control flow obfuscation, including dead code insertion, instruction substitution, opaque predicate, control flow flattening, loop unrolling and subroutine reordering.

3. *O2*: In addition to *O1*, the obfuscator performs data obfuscations including variables encoding, data split and data aggregation. *O2* contains both control and data obfuscations. Therefore, *O2* has a much stronger obfuscation effect that *O1*.

[4]Since this tool is not publicly available, we simulate the approach by BAP's built-in feature to generate DFGs. We implement DFGIsom's normalization rules to simplify DFGs, which are then matched by Ullman's subgraph isomorphism algorithm [109].

We use the source code in OpenSSL as the reference implementation. Since OpenSSL does not include the TEA algorithm, we use the code shown in Wheeler's paper [101] as TEA's reference implementation. First we compile the crypto libraries with different obfuscation options. Then we compile and statically link the 25 collected cryptographic projects to the crypto libraries. At last, we run CryptoHunt and other crypto detection tools to detect them. We evaluate CryptoHunt in two scenarios. First, the testing library is same as the reference library. In this case, we use OpenSSL as both the reference and testing library. The other scenario is that the testing library is different from the reference library. In this case, we use OpenSSL as the reference library and Libgcrypt as the testing library. One exception is that TEA is not included in Libgcrypt, we select another implementation TEA* [112].

6.10.1.4 Evaluation Result

The evaluation result is shown in Table 6.2[5]. Basically, only CryptoHunt is able to detect commonly used cryptographic functions in all cases, while other tools are severely restricted under different obfuscation combinations and algorithm implementations. For example, the advanced dynamic detection tool, Aligot, fails in all of the tasks with the O2 obfuscation option. Next, we provide more details behind the results.

6.10.1.4.1 TEA TEA is a 64-bit cipher which uses 128-bit key. It is usually implemented as 64 rounds of Feistel structure [101]. In CryptoHunt, we use the transformations inside one Feistel structure loop as the reference implementation (see Figure 6.2). As shown in Table 6.2, all tools except Findcrypto2 successfully identify the TEA algorithm in the unobfuscated code in both OpenSSL and Libgcrypt. The reason is Findcrypto2 does not contain TEA's static signature. In the O1 version, DFGIsom fails to detect TEA because data flow graph is obfuscated. Signsrch and CryptoSearcher rely on the magic number `0x9e3779b9` as the static signature. This number cannot be obfuscated by control obfuscation techniques, so Signsrch and CryptoSearcher still work in O1 version. Aligot and Kerchkhoffs are resilient to the control obfuscation techniques. With data obfuscation added in the O2

[5]We find that the crypto detection tools either detect all the five projects in one algorithm category, or detect none of them. Therefore, for simplicity we use the check mark ✓ to indicate that the tool detects all five samples and blank showing it detect none of them.

Table 6.2: Evaluation result on crypto libraries.

Crypto Lib	Algo	Obf	Findcrypto2	Signsrch	CryptoSearcher	DFGIsom	Kerchkhoffs	Aligot	CryptoHunt
OpenSSL	TEA	N		✓	✓	✓	✓	✓	✓
		O1		✓	✓		✓	✓	✓
		O2							✓
	AES	N	✓	✓	✓	✓		✓	✓
		O1	✓	✓	✓			✓	✓
		O2							✓
	RC4	N						✓	✓
		O1						✓	✓
		O2							✓
	MD5	N			✓	✓		✓	✓
		O1			✓			✓	✓
		O2							✓
	RSA	N							✓
		O1							✓
		O2							✓
Libgcrypt	TEA*	N		✓	✓	✓	✓	✓	✓
		O1		✓	✓		✓	✓	✓
		O2							✓
	AES	N	✓	✓		✓			✓
		O1	✓	✓					✓
		O2							✓
	RC4	N						✓	✓
		O1						✓	✓
		O2							✓
	MD5	N			✓	✓		✓	✓
		O1			✓			✓	✓
		O2							✓
	RSA	N							✓
		O1							✓
		O2							✓

version, only CryptoHunt is able to detect the highly obfuscated TEA algorithm. In another implementation TEA*, the result is the same.

6.10.1.4.2 AES The AES design is based on substitution-permutation network [102], which is stronger than the Feistel structure in TEA. We use the core transformation in the innermost loop in OpenSSL's implementation as the reference. Most tools successfully identify AES algorithm in the OpenSSL experiment without obfuscation. We attribute this to AES's distinct feature such as the lookup table. With O1 obfuscation, DFGIsom fails due to the same reason as in TEA. Particularly, we notice that Aligot fails to detect unobfuscated AES algorithm in Ligcrypt when using OpenSSL as the reference. We looked into the source code and binary code and find it is because of the different implementations between OpenSSL and Libgcrypt. The input and out variables in the innermost loop of OpenSSL's implementation is different from those in Libgcrypt. Since Aligot views the loop body as a black box without checking the details inside, it cannot perform more fine-grained detection as CryptoHunt.

6.10.1.4.3 RC4 RC4, a classical steam cipher, generates a random stream of bits as a key stream. The key stream is used to encrypt or decrypt by performing an XOR operation on the input. Typically, the encryption procedure in RC4 is a simple `XOR` operation. It cannot be used as the reference to recognize RC4 algorithm because it will cause lots of false positives. Instead, we use the transformation in the key generation algorithm as the reference implementation. Table 6.2 shows only Aligot and CryptoHunt successfully detect the RC4 algorithm in the unobfuscated program and O1 version. The reason is, unlike TEA and AES, RC4 lacks obvious features that can be used as detection signatures. However, Aligot fails in the O2 version again.

6.10.1.4.4 MD5 MD5 algorithm [103] is a widely used cryptographic hash function to generate message digest. It produces a 128-bit hash value for any input message. The input message is split into chunks of 512-bit and then processed in a main loop. We use the transformations in the main loop as the reference implementation. CryptoSearcher, Aligot, and CryptoHunt successfully detect the clean version of MD5. Typically, there is an initial value for the digest variable

in a MD5 implementation, such as `0x67452301` in OpenSSL. Therefore, CryptoSearcher detects MD5 by searching for this constant value in binaries. Since control obfuscation does not change these constants and the input/output variables in a loop body, CryptoSearcher and Aligot is still able to detect MD5 in O1 option. However, after adding data obfuscation with O2 option, only CryptHunt detects MD5 algorithm.

6.10.1.4.5 RSA The RSA cryptographic algorithm [104] is one of the most widely used public-key cryptosystems. RSA achieves this asymmetric goal based on the computation difficulty of factoring the product of two large prime numbers. Therefore, typically a RSA implementation includes a specific method to represent large integer numbers. Due to the difference between varieties of implementations, this representation can be viewed as an encoding of inputs and outputs. So this "built-in" data encoding makes detection of RSA more difficult than of other cryptographic algorithms. Table 6.2 shows that all of the peer tools fail to identify the RSA algorithm. We find out three reasons contributing to the poor detection result. First, RSA reveals no evident static features and therefore the tools such as CryptoSearcher and Findcrypto2 are not able to detect it. Second, for Aligot, the big number encoding in OpenSSL causes the extracted loop I/O parameters from binary code cannot be directly matched to the reference implementation. In contrast, CryptoHunt takes advantage of bit precise formulas so as to accurately identify the semantically equivalent operations. At last, RSA's modular exponentiation implementation usually contains a main loop which matches the model in Figure 6.3(b). Each iteration of the loop could goes into two branches, either one multiplication or one squaring and a multiplication. Aligot's incomplete loop model causes it to miss the main loop and to fail to detect RSA.

6.10.2 Answer to RQ1: Individual Implementations

In addition to the standard implementation, some cryptographic algorithms allow users to customize some key values to generate a new version. XTEA is such an example. XTEA is the extended version of TEA. One important enhancement is that the number of rounds is not fixed in XTEA, but 64 rounds is suggested. Figure 6.9 shows a reference implementation of the decryption procedure in XTEA. However, malware authors have already abused such flexibility to produce new

```
1  void decipher(uint32_t v[2], uint32_t const key[4],
2                unsigned int num_rounds) {
3    unsigned int i;
4
5    uint32_t v0=v[0], v1=v[1];
6    uint32_t delta=0x9E3779B9, sum=delta*num_rounds;
7
8    for (i=0; i < num_rounds; i++) {
9      v1 -= (((v0<<4)^(v0>>5))+v0) ^ (sum+key[(sum>>11) & 3]);
10     sum -= delta;
11     v0 -= (((v1<<4)^(v1>>5))+v1) ^ (sum+key[sum & 3]);
12   }
13   v[0] = v0; v[1] = v1;
14 }
```

Figure 6.9: A reference implementation of XTEA's decryption.

```
1  unsigned int num_rounds = 11, i;
2
3  uint32_t v0, v1;
4  uint32_t delta = 0x61C88647, sum = 0xCC623AF3;
5
6  for (i=0; i < num_rounds; i++) {
7    v1 -= (((v0<<4)^(v0>>5))+v0) ^ (sum+key[(sum>>11) & 3]);
8    sum -= delta;
9    v0 -= (((v1<<4)^(v1>>5))+v1) ^ (sum+key[sum & 3]);
10 }
```

Figure 6.10: The decryption function in an Apache Module injection malware.

variations to evade detection. A recent study [60] reports that a variant of XTEA is used in an Apache module injection attack. We reverse engineer the Apache module's binary code and manually recover the new XTEA version. Figure 6.10 presents the core part of the new XTEA in C code.

In Figure 6.10, we can observe that the malware author modified XTEA algorithm by replacing the original magic number `0x9E3779B9` with `0x61C88647`. He also used 11 rounds of transformation rather than the suggested 64 rounds. In order to show whether CryptoHunt can detect this modified version of XTEA, we implement the function in Figure 6.10 as a C program. The source code shown in

Table 6.3: Evaluation result on an XTEA variant from malware.

Algo	Obf	Findcrypto2	Signsrch	CryptoSearcher	DFGIsom	Kerchkhoffs	Aligot	CryptoHunt
Modified XTEA	N				✓			✓
	O1							✓
	O2							✓

Figure 6.9 is used as the reference implementation. Similar to the evaluation on crypto libraries, we compile the testing program with different obfuscation options *N*, *O1*, and *O2*. We also run other detection tools to compare with CryptoHunt. The result is shown in Table 6.3.

From the result, we can see that only CryptoHunt detected the modified XTEA in all three versions. Because the malware author changes the magic number and rounds, all static tools based on these signatures fail to detect it. Particularly, due to the new magic number, the computation in the loop body changes. Therefore, input and output values in the modified version do not match the reference implementation, which causes Aligot to deliver a poor detection result. DFGIsom correctly extracts and match the DFG so it can identify the modified XTEA in the unobfuscated version. This case study shows that CryptoHunt is able to catch the crucial transformations related to cryptographic functions and ignore the differences introduced by obfuscation and modification to the original algorithm.

6.10.3 Answer to RQ1: Malware Samples

Table 6.4 shows the evaluation results on malware samples we collect from the Internet, including now-infamous crypto-ransomware. RansomCrypt is a ransomware sample. When first run on a system, it iterates all files and encrypts them using TEA. Another ransomware sample, Locky, utilizes AES to encrypt files in victim's computer. Sality malware code has two sections; the first section decrypts the second section using RC4 and redirects the execution to the beginning of the second section. Waledac malware sample runs MD5 to generate a unique ID for every

Table 6.4: Evaluation result on malware samples.

Malware	Algo	Findcrypto2	Signsrch	CryptoSearcher	DFGIsom	Kerchkhoffs	Aligot	CryptoHunt
RansomCrypt	TEA		✓	✓		✓	✓	✓
Locky	AES							✓
Sality	RC4						✓	✓
Waledac	MD5						✓	✓
CryptoWall	RSA							✓

Table 6.5: False positive evaluation dataset.

Category	Programs
Core Utilities	ls, cp, mv, cat, head
Compression tools	Gzip, bzip2, 7-zip
Server	thttpd, lighttpd

bot. The notorious CryptoWall ransomware encrypts a wide variety of files in the compromised computer using RSA.

From Table 6.4, we can see that many detection tools are able to identify TEA in RansomCrypt. It's because RansomCrypt uses the standard TEA algorithm with only slight obfuscation. However, in other crypto algorithms, most of the detection tools fail. One reason is that usually malware authors call Windows Crypto API in the malware and apply obfuscation methods to hide the API call address. The malware itself does not include the crypto algorithm implementation. Therefore, signature-based tools fail to detect the crypto algorithms. Aligot fails to detect AES in Locky due to the implementation difference between OpenSSL and Windows crypto API. It also fails to detect RSA for the similar reasons we discussed in Section 6.10.1.4: 1) big-integer encoding; 2) incomplete loop identification.

6.10.4 Answer to RQ2: Normal Programs

Too many false positives limit cryptographic function detection's application in practice. In this section, we test CryptoHunt with a set of normal programs to evaluate its false positives. As shown in Table 6.5, our dataset includes GNU

Table 6.6: CryptoHunt's offline analysis performance on OpenSSL.

Time (min)	TEA	AES	RC4	MD5	RSA
Loop identification	12.7	23.1	21.5	24.8	33.2
Variable mapping	0.7	2.6	1.9	2.1	3.4
Verification	2.3	4.1	4.5	5.2	6.7
Total	15.7	30.8	27.9	32.1	43.3

core utilities, compression tools, and lightweight server programs. We choose compression tools because they also contain intensive bitwise operations, and server programs contain a large number of loops. Our test dataset includes two groups. The first is the original programs without any change. In the other group, we inject magic numbers which could be used by crypto detection tools such as `0x9E3779B9`. The second group mimics possible malware attacks. They are likely to insert known signatures into benign programs to mislead detection tools. However, our result shows that CryptoHunt reports no cryptographic function detected in all cases. That means CryptoHunt has no false positive in our test dataset.

6.10.5 Answer to RQ3: Overall Performance

In this section, we provide the answer to RQ3 about CryptoHunt's performance. There are two phases when analyzing using CryptoHunt, trace logging and offline analysis. We take advantage of Pin [85] to record the execution trace. The online trace logging overhead is typically 5-6X slowdown. Table 6.6 presents the offline analysis performance of CryptoHunt. We record the running time of different components in CryptoHunt. The most time-consuming part is loop identification since it goes over the whole trace multiple times and tries to identify nested loops. Based on our observation, the nested loop level significantly increases the loop identification time. Another factor that affects the performance is the number of input and output variables of the loops. More variables will cause the variable mapping algorithm generate more mapping candidates, which potentially raise the chance of launching a theorem prover. Compared with Aligot's result, which usually takes more than 6 hours to analyze one execution trace, CryptoHunt delivers much better performance.

Table 6.7: Evaluation of the Mapping Algorithm. The second column shows the number of loops. The third column shows the number of mapping variable candidates. "NM" stands for "No mapping", which means the number of STP queries without the mapping algorithm. Similarly, the column "M" shows the number of STP queries with the mapping algorithm.

Algorithms	Loops	Vars	# of STP Queries		
			NM	M	Ratio (%)
TEA	7	41	2825	173	6.1
AES	13	96	7138	351	4.9
RC4	9	73	6055	337	5.6
MD5	8	77	8301	429	5.2
RSA	15	89	15521	803	5.2

6.10.6 Answer to RQ3: Mapping Algorithm

In this section, we present the experimental data to show the performance improvement introduced by our guided fuzzing approach, which aims to reduce the number of symbolic variables to be verified by a theorem solver. The data is collected in the OpenSSL evaluation in Table 6.2. We collect the number of identified loops, number of mapping variables candidates, and number of STP queries. The result is shown in Table 6.7. In order to compare with the mapping algorithm, we also implement a naive mapping procedure, which generates every possible combination. However, the naive mapping outputs too many candidates. Thus we add some simple heuristics to reduce the candidate number. The reduced number is shown in the "NM" row. From the data, we can see that our mapping algorithm reduce about 95% STP queries on average.

Algorithm 2 Mapping I/O Variables

Parameters:
  $\vec{u}^T = F_1(\vec{x}^T), \vec{v}^T = F_2(\vec{y}^T)$: Boolean formulas
  L: Current partial mapping list.
  M_{in}: Full mapping of input.
  M_{out}: Full mapping of output.
function VarMapping(F_1, F_2, L, M_{in}, M_{out})
  if L is empty **then**
    return (M_{in}, M_{out})
  end if
  $I \leftarrow$ CreateIdentityMatrix(L)
  $R \leftarrow$ RandomInput(M_{in})
  $M_1^I \leftarrow$ CreateInputMatrix(I, R, F_1)
  $M_2^I \leftarrow$ CreateInputMatrix(I, R, F_2)
  $M_1^O \leftarrow$ CreateOM(F_1, M_1^I)
  $M_2^O \leftarrow$ CreateOM(F_2, M_2^I)
  $r\vec{v}_1 \leftarrow$ CreateRowVector(M_1^O)
  $c\vec{v}_1 \leftarrow$ CreateColumnVector(M_1^O)
  $r\vec{v}_2 \leftarrow$ CreateRowVector(M_2^O)
  $c\vec{v}_2 \leftarrow$ CreateColumnVector(M_2^O)
  if Sort($r\vec{v}_1$) $\neq$ Sort($r\vec{v}_2$) || Sort($c\vec{v}_1$) $\neq$ Sort($c\vec{v}_2$) **then**
    return False
  end if
  UpdateMapping($L, r\vec{v}_1, r\vec{v}_2$)
  UpdateMapping($L, c\vec{v}_1, c\vec{v}_2$)
  Reduce(L)
  if L is not a partial mapping **then**
    return False
  end if
  if $\exists s \mapsto t \in L$ **then**
    Remove $s \mapsto t$ from L
    Add $s \mapsto t$ to M_{in} or M_{out}
    VarMapping(F_1, F_2, L, M_{in}, M_{out})
  else
    for Permute the set connection $s' \mapsto t' \in L$ **do**
      Remove $s' \mapsto t'$ from L
      Add the permutation to M_{in} or M_{out}
      VarMapping(F_1, F_2, L, M_{in}, M_{out})
    end for
  end if
end function

Chapter 7
Discussion and Future Work

Opaque predicate detection techniques such as LOOP, rely on solvers to solve path constraints. However, path conditions may include arithmetic constraints that are infeasible to solve. For example, the constraints are computationally unsolvable, or they are undecidable. Those path conditions present challenges to program analysis methods such as symbolic execution plus constraint solver. In this chapter, we list several predicates that is hard to analysis and propose possible future research work. Moreover, we also discuss the limitations and future work in the cryptographic function detection research.

7.1 Complex Path Conditions

In general, non-linear integer constraints are undecidable [113]. Therefore, these constraints cause solvers unable to solve the path condition. Existing classical concolic methods mitigate these problems by replacing some symbolic values with concrete values using heuristic strategies. These reduction methods slightly improve the solver's ability so that they can handle non-linear predicates in some cases. However, usually the simplification leads to over-approximation and misses possible solutions.

Dinges and Agha [114] present a method combining symbolic reasoning, concrete evaluation, and heuristic search. For non-linear constraints, it finds a point induced by the linear constraints with an solver. Then starting from this point, it uses adaptive search guided by the constraint fitness function to find a solution. Li et al. [115] propose a new symbolic execution method. Instead of relying on the classical constraint solving, the feasibility problem is first transformed into

optimization problems by minimizing some dissatisfaction degree. The optimization problem are then handled by a optimization solver through machine learning guided sampling and validation [116]. However, in general these work are based on heuristic assumptions or sampling so they still suffer from the over-approximation limitation.

7.2 Inductive Constraints

Another category of constraints that are hard for solvers to solve is inductive constraints. Existing SMT solvers such as Z3 and CVC are not able to solve statements that requiring non-trivial use of induction. Leino [117] presents a pre-processing of formula to perform inductive reasoning in program verifier. A more recent work proposed by Reynolds [118] enhances SMT solvers to handle inductive constraints. It uses heuristics to filter irrelevant intermediate lemmas when proving the inductive constraints.

7.3 Unsolved Conjectures

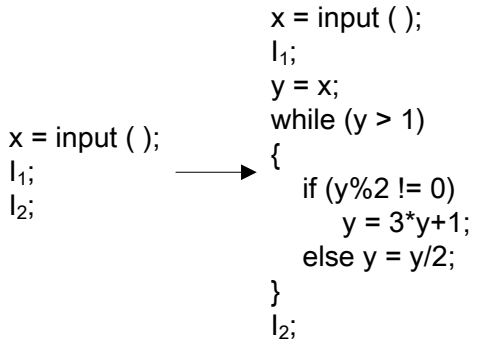


Figure 7.1: Example of 3x+1 conjecture.

Recently Wang et al. [119] proposed a more stealthy obfuscation scheme by incorporating linear unsolved conjectures, which appear to be correct but without proof. Figure 7.1 presents an example of embedding the well-known 3x+1 conjecture into a program. This conjecture asserts that given any positive integer y, the loop will always terminate. In principle, we could treat the conjunction of branch conditions derived from the unroll loop as a single opaque predicate, which always

evaluates to be true for any positive integer. However, different from dynamic opaque predicates, the number of conditions is various under different inputs and conditions themselves are not correlated as well. To detect such unsolved conjectures, we observe that all the examples [119] will eventually converge to a fixed value regardless of the initial value. Therefore, one naive solution is to automatically generate test cases to explore different paths and observe whether the multiple inputs cover the same value when the conjecture loop ends.

7.4 Cryptographic Function Detection

Since CryptoHunt works with adversaries, we have to consider how a skilled attacker could circumvent CryptoHunt once our approach is known. In this section, we discuss CryptoHunt's limitations, possible attacks, and countermeasures, which also light up our future work. First, like any binary dynamic analysis approach, one limitation of CryptoHunt is its incomplete path coverage. Typically, CryptoHunt can detect cryptographic functions exhibiting during run time. One way to increase the path coverage is to leverage automatic input generation techniques [62, 120]. The static analysis may consider multiple paths. However, the various obfuscation methods adopted by malware authors will undoubtedly impede the accurate static analysis [5]. The best way to reconcile such tradeoff is still a hot subject of research in security analysis. We believe CryptoHunt is practical in analyzing obfuscated malware.

Second, our prototype is not well optimized for performance. For example, CryptoHunt's online logging imposes 5-6X slowdown on average. We can rely on pervasive multi-core architectures to parallelize dynamic instrumentation [121] for better runtime performance. Meanwhile, the performance of CryptoHunt's offline analysis depends on the trace size. The loop detection will become performance bottleneck when the trace size is too large. We leave addressing the performance issue as our future work.

Another threat to CryptoHunt is environment-sensitive malware [122–124]. Since we run malware with Pin, a malware sample can detect itself running in Pin instead of the physical machine and then quit immediately. A possible countermeasure to such sandbox environment check is analyzing malware in a transparent analysis platform via hardware virtualization (e.g., Ether [125]).

Our symbolic variable mapping depends on the output of our backward slicing, which already filters out irrelevant instructions. However, attackers can defeat it by adding artificial dependencies between normal data flow and redundant code. In an extreme case, the sliced segment could contain all the executed instructions. While such an attack could reduce the efficacy of CryptoHunt, at the same time it also requires extensive efforts and high cost for attackers. In summary, CryptoHunt significantly raises the bar for skilled cybercriminals to defeat our approach.

Chapter 8
Conclusion

Opaque predicates have been widely used in software protection and malicious program to obfuscation program control flow. Existing efforts to detect opaque predicates are either heuristics-based or work only on specific categories. Dynamic opaque predicate obfuscation is regarded as a promising method since the predicate values may vary in different executions and thus make them more resilient to detection. Furthermore, cryptographic functions are adopted to help hide opaque predicates from detection. The attack and defense techniques related to opaque predicates attract many researchers' attention.

In this thesis, we present our new research work related to opaque predicate obfuscation and deobfuscation in binary code. First, we propose LOOP, a program logic-based and obfuscation resilient approach to opaque predicate detection in binary code. Our approach represents the characteristics of various opaque predicates with logical formulas and verifies them with a constraint solver. LOOP detects not only simple invariant opaque predicates, but also advanced contextual and dynamic opaque predicates. Our experimental results show that LOOP is effective in detecting opaque predicates in a range of benign and obfuscated binary programs. By diagnosing culprit branches derived from opaque predicates in an execution trace, LOOP can help analysts for further deobfuscation. The experiment of speeding up code normalization for matching metamorphic malware variants confirms the value of LOOP in malware defenses.

Second, an enhanced control flow obfuscation method called generalized dynamic opaque predicate is proposed. Our method automatically inserts dynamic opaque predicates into common program structures and is hard to be detected by the state-of-the-art formal program semantics-based deobfuscation tools. The experimental

results show the efficacy and resilience of our method with negligible performance overhead.

Third, a new technique to detect cryptographic function in binary code is proposed for defense against the opaque predicates enhanced by cryptography. Our technique is called *bit-precise symbolic loop mapping*. It first captures the specific features of cryptographic algorithms with boolean formulas, which are later used as signatures for efficiently matching possible cryptographic algorithms in obfuscated binary code. We have implemented our approach called CryptoHunt and evaluated it with a set of cryptographic algorithms under different obfuscation schemes and combinations. Our comparative experiments show that CryptoHunt outperforms existing work in terms of better obfuscation resilience and broader detection scope.

Appendix
Publication List

1. Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu, BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *Proceedings of the 26th USENIX Security Symposium*, Vancouver, BC, Canada, August 16-18, 2017.

2. Dongpeng Xu, Jiang Ming, and Dinghao Wu, Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*, San Jose, CA, May 22-24, 2017.

3. Dongpeng Xu, Jiang Ming, and Dinghao Wu, Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method. In *Proceedings of The 19th Information Security Conference*, Honolulu, HI, USA, September 7-9, 2016.

4. Jiang Ming, Dongpeng Xu, and Dinghao Wu, MalwareHunt: Semantics-Based Malware Diffing Speedup by Normalized Basic Block Memoization. *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 3, August 2017.

5. Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu, LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, Denver, Colorado, USA, October 12-16, 2015.

6. Jiang Ming, Dongpeng Xu, and Dinghao Wu, Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference. In *Proceedings*

of the 30th IFIP SEC 2015 International Information Security and Privacy Conference (IFIP SEC 2015), Hamburg, Germany, May 26-28, 2015.

Bibliography

[1] Collberg, C., C. Thomborson, and D. Low (1997) *A taxonomy of obfuscating transformations*, *Tech. rep.*, The University of Auckland.

[2] ——— (1998) "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'98)*.

[3] Collberg, C. and J. Nagra (2009) *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*, Addison-Wesley Professional.

[4] Linn, C. and S. Debray (2003) "Obfuscation of executable code to improve resistance to static disassembly," in *CCS'03*.

[5] Moser, A., C. Kruegel, and E. Kirda (December 2007) "Limits of static analysis for malware detection," in *Proceedings of the 23th Annual Computer Security Applications Conference (ACSAC'07)*.

[6] Popov, I. V., S. K. Debray, and G. R. Andrews (2007) "Binary obfuscation using signals," in *USENIX Security '07*.

[7] Sharif, M., A. Lanzi, J. Giffin, and W. Lee (2008) "Impeding Malware Analysis Using Conditional Code Obfuscation," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*.

[8] Wu, Z., 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 securit (CCS'10)*.

[9] Schiffman, M. (2010) "A brief history of malware obfuscation," *Cisco, security, http://blogs. cisco. com/security*.

[10] Wong, W. and M. Stamp (2006) "Hunting for metamorphic engines," *Computer Virology*, **2**(3), pp. 211–229.

[11] Konstantinou, E. and S. Wolthusen (2008) *Metamorphic Virus: Analysis and Detection*, *Tech. rep.*, RHUL-MA-2008-02, University of London.

[12] Christodorescu, M. and S. Jha (2003) "Static Analysis of Executables to Detect Malicious Patterns," in *Proceedings of the 12th conference on USENIX Security Symposium*.

[13] Wang, C., J. Hill, J. C. Knight, and J. W. Davidson (2001) "Protection of Software-Based Survivability Mechanisms," in *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN'01)*.

[14] Roundy, K. A. and B. P. Miller (2013) "Binary-code Obfuscations in Prevalent Packer Tools," *ACM Computing Surveys*, **46**(1).

[15] Cappaert, J. and B. Preneel (2010) "A General Model for Hiding Control Flow," in *Proceedings of the 10th Annual ACM Workshop on Digital Rights Management (DRM'10)*.

[16] Chen, H., L. Yuan, X. Wu, B. Zang, B. Huang, and P.-c. Yew (2009) "Control Flow Obfuscation with Information Flow Tracking," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*.

[17] Coppens, B., B. De Sutter, and J. Maebe (2013) "Feedback-driven Binary Code Diversification," *ACM Transactions on Architecture and Code Optimization (TACO)*, **9**(4).

[18] Larsen, P., A. Homescu, S. Brunthaler, and M. Franz (2014) "SoK: Automated Software Diversity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP'14)*.

[19] Bruschi, D., L. Martignoni, and M. Monga (2006) "Detecting Self-mutating Malware Using Control-Flow Graph Matching," in *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'06)*.

[20] ——— (2007) "Code Normalization for Self-Mutating Malware," *IEEE Security and Privacy*, **5**(2).

[21] Arboit, G. (2002) "A Method for Watermarking Java Programs via Opaque Predicates," in *Proceedings of 5th International Conference on Electronic Commerce Research (ICECR-5)*.

[22] Myles, G. and C. Collberg (2006) "Software watermarking via opaque predicates: Implementation, analysis, and attacks," *Electronic Commerce Research*, **6**(2), pp. 155 – 171.

[23] Kovacheva, A. (2013) *Efficient Code Obfuscation for Android*, Master's thesis, University of Luxembourg.

[24] Collberg, C., G. Myles, and A. Huntwork (2003) "Sandmark–A Tool for Software Protection Research," *IEEE Security and Privacy*, **1**(4), pp. 40–49.

[25] Junod, P., J. Rinaldini, J. Wehrli, and J. Michielin (2015) "Obfuscator-LLVM - Software Protection for the Masses," in *Proceedings of the 1st International Workshop on Software PROtection (SPRO'15)*.

[26] Madou, M., L. Van Put, and K. De Bosschere (2006) "LOCO: An Interactive Code (De)Obfuscation Tool," in *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'06)*.

[27] Anckaert, B., M. Madou, B. D. Sutter, B. D. Bus, K. D. Bosschere, and B. Preneel (2007) "Program obfuscation: a quantitative approach," in *Proceedings of the 2007 ACM workshop on Quality of Protection (QoP'07)*.

[28] Szor, P. (2005) *The Art of Computer Virus Research and Defense*, Addison-Wesley Professional.

[29] DefenseCode (last reviewed, 04/27/2015), "Diving into recent 0day Javascript obfuscations," `http://blog.defensecode.com/2012/10/diving-into-recent-0day-javascript.html`.

[30] Zobernig, L., S. D. Galbraith, and G. Russello "Indistinguishable Predicates: A New Tool for Obfuscation," .

[31] Madou, M. (2007) *Application Security through Program Obfuscation*, Ph.D. thesis, Ghent University.

[32] Preda, M. D., M. Madou, K. D. Bosschere, and R. Giacobazzi (2006) "Opaque Predicate Detection by Abstract Interpretation," in *Proceedings of 11th International Conference on Algebriac Methodology and Software Technology (AMAST'06)*.

[33] Quyn, N. A. (2013) "OptiCode: Machine Code Deobfuscation for Malware Analysis," in *Proceedings of the 2013 SyScan*.

[34] Udupa, S. K., S. K. Debray, and M. Madou (2005) "Deobfuscation: Reverse Engineering Obfuscated Code," in *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05)*.

[35] Brumley, D. and J. Newsome (2006) *Alias Analysis for Assembly*, *Tech. Rep. CMU-CS-06-180R*, School of Computer Science, Carnegie Mellon University.

[36] Ming, J., M. Pan, and D. Gao (2012) "iBinHunt: Binary Hunting with Inter-Procedural Control Flow," in *Proceedings of the 15th Annual International Conference on Information Security and Cryptology (ICISC'12)*.

[37] Brumley, D., I. Jager, T. Avgerinos, and E. J. Schwartz (2011) "BAP: A Binary Analysis Platform," in *Proceedings of the 23rd international conference on computer aided verification (CAV'11)*.

[38] Lattner, C. and V. Adve (2004) "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*.

[39] OKane, P., S. Sezer, and K. McLaughlin (2011) "Obfuscation: The Hidden Malware," *IEEE Security and Privacy*, **9**(5).

[40] Wang, C., J. Davidson, J. Hill, and J. Knight (2001) "Protection of software-based survivability mechanisms," in *Proceedings of International Conference on Dependable Systems and Networks (DSN'01)*.

[41] Collberg, C. and J. Nagra (2009) *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, chap. 4.4, Addison-Wesley Professional, pp. 258–276.

[42] Sikorski, M. and A. Honig (2012) *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, chap. 13, No Starch Press, pp. 269–296.

[43] Viticchié, A., L. Regano, M. Torchiano, C. Basile, M. Ceccato, P. Tonella, and R. Tiella (2016) "Assessment of Source Code Obfuscation Techniques," in *Proceedings of the 16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'16)*.

[44] Lin, Z., 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 (NDSS'10)*.

[45] Lee, J., T. Avgerinos, and D. Brumley (2011) "TIE: Principled Reverse Engineering of Types in Binary Programs," in *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS'11)*.

[46] Drape, S. (2010) *Intellectual Property Protection using Obfuscation*, *Tech. Rep. RR-10-02*, Oxford University Computing Laboratory.

[47] Palsberg, J., S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang (2000) "Experience with Software Watermarking," in *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC'00)*.

[48] Bodík, R., R. Gupta, and M. L. Soffa (1997) "Refining Data Flow Information Using Infeasible Paths," in *Proceedings of the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'97)*.

[49] Ngo, M. N. and H. B. K. Tan (2007) "Detecting Large Number of Infeasible Paths Through Recognizing Their Patterns," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'07)*.

[50] Bueno, P. M. S. and M. Jino (2000) "Identification of Potentially Infeasible Program Paths by Monitoring the Search for Test Data," in *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00)*.

[51] Lutz, N. (2008) "Towards revealing attacker's intent by automatically decrypting network traffic," *Mémoire de maitrise, ETH Zürich, Switzerland*.

[52] Wang, Z., X. Jiang, W. Cui, X. Wang, and M. Grace (2009) "ReFormat: Automatic Reverse Engineering of Encrypted Messages," in *Proceedings of the 14th European Conference on Research in Computer Security (ESORICS'09)*.

[53] Matenaar, F., A. Wichmann, F. Leder, and E. Gerhards-Padilla (2012) "CIS: The Crypto Intelligence System for Automatic Detection and Localization of Cryptographic Functions in Current Malware," in *Proceedings of the 7th International Conference on Malicious and Unwanted Software (MALWARE'12)*.

[54] Lestringant, P., F. Guihéry, and P.-A. Fouque (2015) "Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS'15)*.

[55] Calvet, J., J. M. Fernandez, and J.-Y. Marion (2012) "Aligot: Cryptographic Function Identification in Obfuscated Binary Programs," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*.

[56] Li, X., X. Wang, and W. Chang (2014) "CipherXRay: Exposing Cryptographic Operations and Transient Secrets from Monitored Binary Execution," *IEEE Transactions on Dependable and Secure Computing*, **11**(2).

[57] Gröbert, F., C. Willems, and T. Holz (2011) "Automated Identification of Cryptographic Primitives in Binary Programs," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID'11)*.

[58] Zhao, R., D. Gu, J. Li, and R. Yu (2011) "Detection and Analysis of Cryptographic Data Inside Software," in *Proceedings of the 14th International Conference on Information Security (ISC'11)*.

[59] Schmitt, P., "A Different Kind of Crypto: Crypto Algorithms Designed for Payload Obfuscation," BlackHat 2014.

[60] Grunzweig, J. (2013), "Digging Into the New Apache Injection Module," `https://www.trustwave.com/Resources/SpiderLabs-Blog/Digging-Into-the-New-Apache-Injection-Module/`, SpiderLabs Blog.

[61] King, J. C. (1976) "Symbolic Execution and Program Testing," *Commun. ACM*, **19**(7), pp. 385–394.

[62] Godefroid, P., M. Y. Levin, and D. Molnar. (2008) "Automated Whitebox Fuzz Testing," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*.

[63] Vanegue, J., S. Heelan, and R. Rolles (2012) "SMT Solvers for Software Security," in *Proceedings of the 6th USENIX Conference on Offensive Technologies (WOOT'12)*.

[64] Bounimova, E., P. Godefroid, and D. Molnar (2013) "Billions and Billions of Constraints: Whitebox Fuzz Testing in Production," in *Proceedings of the International Conference on Software Engineering (ICSE'13)*.

[65] Cadar, C., V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. (2006) "EXE: Automatically generating inputs of death," in *Proceedings of the 2006 ACM Conference on Computer and Communications Security (CCS'06)*.

[66] Cadar, C., D. Dunbar, and D. Engler. (2008) "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 2008 USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*.

[67] Song, D., D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena (2008) "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *4th International Conference on Information Systems Security. Keynote invited paper*.

[68] Godefroid, P., N. Klarlund, and K. Sen (2005) "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*.

[69] Brumley, D., J. Caballero, Z. Liang, J. Newsome, and D. Song (2007) "Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation," in *Proceedings of 16th USENIX Security Symposium*.

[70] Banerjee, A., A. Roychoudhury, J. A. Harlie, and Z. Liang (2010) "Golden Implementation Driven Software Debugging," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*.

[71] Kang, M. G., S. McCamant, P. Poosankam, and D. Song (2011) "DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*.

[72] Ming, J., D. Xu, and D. Wu (2015) "Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference," in *Proceedings of the 30th IFIP SEC 2015 International Information Security and Privacy Conference (IFIP SEC'15)*.

[73] Gao, D., M. Reiter, and D. Song (2008) "BinHunt: Automatically finding semantic differences in binary programs," in *Proceedings of the 10th International Conference on Information and Communications Security (ICICS'08)*.

[74] Luo, L., J. Ming, D. Wu, P. Liu, and S. Zhu (2014) "Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*.

[75] Egele, M., M. Woo, P. Chapman, and D. Brumley (2014) "Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components," in *23rd USENIX Security Symposium (USENIX Security'14)*.

[76] Pewny, J., B. Garmany, R. Gawlik, C. Rossow, and T. Holz (2015) "Cross-Architecture Bug Search in Binary Executables," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*.

[77] Chandramohan, M., Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and T. H. B. Kuan (2016) "BinGo: Cross-Architecture Cross-OS Binary Search," in *Proceedings of the 2016 ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'16)*.

[78] Ming, J., D. Xu, and D. Wu (2016) "MalwareHunt: semantics-based malware diffing speedup by normalized basic block memoization," *Journal of Computer Virology and Hacking Techniques*.

[79] Luo, L., J. Ming, D. Wu, P. Liu, and S. Zhu (2017) "Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection," *IEEE Transactions on Software Engineering*.

[80] Ming, J., D. Xu, L. Wang, and D. Wu (2015) "LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*.

[81] Liu, L., J. Ming, Z. Wang, D. Gao, and C. Jia (2009) "Denial-of-Service Attacks on Host-Based Generic Unpackers," in *Proceedings of the 11th International Conference on Information and Communications Security (ICICS'09)*.

[82] Agrawal, H. and J. R. Horgan (1990) "Dynamic Program Slicing," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, ACM, pp. 246–256.

[83] Ganesh, V. and D. L. Dill (2007) "A Decision Procedure for Bit-vectors and Arrays," in *Proceedings of the 2007 International Conference in Computer Aided Verification (CAV'07)*.

[84] Moura, L. D. and N. Bjørner (2008) "Z3: an efficient SMT solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

[85] Luk, C.-K., R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood (2005) "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*.

[86] Coogan, K., G. Lu, and S. Debray (2011) "Deobfuscation of Virtualization-Obfuscated Software," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*.

[87] Cordella, L., P. Foggia, C. Sansone, and M. Vento (2004) "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **26**(10), pp. 1367–1372.

[88] Linn, C. 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 (CCS'03)*.

[89] Wang, S., P. Wang, and D. Wu (2015) "Reassembleable Disassembling," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*.

[90] Zhang, F., D. Wu, P. Liu, and S. Zhu (2014) "Program Logic Based Software Plagiarism Detection," in *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering (ISSRE'14)*.

[91] Xu, D., J. Ming, and D. Wu (2016) "Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method," in *Proceedings of The 19th Information Security Conference (ISC'16)*, pp. 323–342.

[92] Hind, M. and A. Pioli (2000) "Which pointer analysis should I use?" in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00)*, ACM, pp. 113–123.

[93] Conte, S. D., H. E. Dunsmore, and V. Y. Shen (1986) *Software engineering metrics and models*, Benjamin-Cummings Publishing Co., Inc.

[94] Ming, J., D. Xu, L. Wang, and D. Wu (2015) "LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*.

[95] Xu, D., J. Ming, and D. Wu (2017) "Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping," in *Proceddings of The 38th IEEE Symposium on Security and Privacy (SP'17)*, pp. 921–937.

[96] Martignoni, L., M. Christodorescu, and S. Jha (2007) "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware," in *Proceedings of the 2007 Annual Computer Security Applications Conference(ACSAC'07)*.

[97] Necula, G. C., S. McPeak, S. P. Rahul, and W. Weimer (2002) "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*.

[98] Royal, P., M. Halpin, D. Dagon, R. Edmonds, and W. Lee (2006) "Polyunpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," in *Proceedings of 2006 Annual Computer Security Applications Conference (ACSAC)*, IEEE Computer Society, pp. 289–300.

[99] Caballero, J., P. Poosankam, C. Kreibich, and D. Song (2009) "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*.

[100] Zhu, W., C. Thomborson, and F.-Y. Wang (2006) "Applications of Homomorphic Functions to Software Obfuscation," in *Proceedings of the 2006 International Workshop on Intelligence and Security Informatics (WISI'06)*.

[101] Wheeler, D. J. and R. M. Needham (1994) "TEA, a tiny encryption algorithm," in *Fast Software Encryption*, Springer, pp. 363–366.

[102] Daemen, J. and V. Rijmen (2013) *The design of Rijndael: AES–the advanced encryption standard*, Springer Science & Business Media.

[103] Rivest, R., "The MD5 Message-Digest Algorithm," `http://www.rfc-base.org/txt/rfc-1321.txt`.

[104] Rivest, R. L., A. Shamir, and L. Adleman (1978) "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, **21**(2), pp. 120–126.

[105] Chun (2004), "x3chun's CryptoSearcher," `http://x3chun.reteam.org/`.

[106] Guilfanov, I. (2015), "IDA-Pro/plugins/FindCrypt2," `https://www.aldeid.com/wiki/IDA-Pro/plugins/FindCrypt2`, aldeid.

[107] Auriemma, L., "Signsrch Tool," `http://aluigi.altervista.org/mytoolz.htm`, tool for searching signatures inside files.

[108] Eagle, C. (2011) *The IDA pro book: the unofficial guide to the world's most popular disassembler*, No Starch Press.

[109] Ullmann, J. R. (1976) "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, **23**(1), pp. 31–42.

[110] Junod, P., J. Rinaldini, J. Wehrli, and J. Michielin (2015) "Obfuscator-LLVM – Software Protection for the Masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection (SPRO'15)*.

[111] You, I. and K. Yim (2010) "Malware obfuscation techniques: A brief survey," in *Proceedings of the 2010 International Conference on Broadband, Wireless Computing, Communication and Applications*.

[112] Williams, D. (2008) "The Tiny Encryption Algorithm (TEA)," *Network Security*, pp. 1–14.

[113] Davis, M. (1973) "Hilbert's tenth problem is unsolvable," *The American Mathematical Monthly*, **80**(3), pp. 233–269.

[114] Dinges, P. and G. Agha (2014) "Solving complex path conditions through heuristic search on induced polytopes," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, pp. 425–436.

[115] Li, X., Y. Liang, H. Qian, Y.-Q. Hu, L. Bu, Y. Yu, X. Chen, and X. Li (2016) "Symbolic execution of complex program driven by machine learning based constraint solving," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ACM, pp. 554–559.

[116] Yu, Y., H. Qian, and Y.-Q. Hu (2016) "Derivative-free optimization via classification," in *Thirtieth AAAI Conference on Artificial Intelligence*.

[117] Leino, K. R. M. (2012) "Automating induction with an SMT solver," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, Springer, pp. 315–331.

[118] Reynolds, A. and V. Kuncak (2015) "Induction for SMT solvers," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, Springer, pp. 80–98.

[119] Wang, Z., J. Ming, C. Jia, and D. Gao (2011) "Linear Obfuscation to Combat Symbolic Execution," in *Proceedings of the 2011 European Symposium on Research in Computer Security (ESORICS'11)*.

[120] Moser, A., C. Kruegel, and E. Kirda (2007) "Exploring multiple execution paths for malware analysis," in *Proceedings of the 2007 IEEE Symposium of Security and Privacy*.

[121] Zhao, Q., I. Cutcutache, and W.-F. Wong (2010) "PiPA: Pipelined Profiling and Analysis on Multicore Systems," *ACM Transactions on Architecture and Code Optimization*, **7**(3).

[122] Kirat, D. and G. Vigna (2015) "MalGene: Automatic Extraction of Malware Analysis Evasion Signature," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*.

[123] Lindorfer, M., C. Kolbitsch, and P. M. Comparetti (September 2011) "Detecting Environment-Sensitive Malware," in *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID'11), Menlo Park, CA, USA*.

[124] Kirat, D., G. Vigna, and C. Kruegel (2014) "BareCloud: Bare-metal Analysis-based Evasive Malware Detection," in *Proceedings of the 23rd USENIX Conference on Security Symposium.*

[125] Dinaburg, A., P. Royal, M. Sharif, and W. Lee (2008) "Ether: Malware Analysis via Hardware Virtualization Extensions," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS'08).*

Vita

Dongpeng Xu

Dongpeng Xu is currently a Ph.D. candidate in the College of Information Sciences and Technology of the Pennsylvania State University, where he is a member of the Software System Security Research Lab. His research interest is software security, especially program analysis on binary code, malware analysis and detection, software protection, software testing, program similarity analysis, and model checking. He received the B.S. degree in Fashion Design and Engineering from Jilin University in 2009 and the M.E. degree in Software Engineering from University of Science and Technology of China in 2013.