Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping

Dongpeng Xu
The Pennsylvania State University
University Park, USA
Email: dux103@ist.psu.edu

Jiang Ming
The University of Texas at Arlington
Arlington, USA
Email: jiang.ming@uta.edu

Dinghao Wu
The Pennsylvania State University
University Park, USA
Email: dwu@ist.psu.edu

***Abstract*—Cryptographic functions have been commonly abused by malware developers to hide malicious behaviors, disguise destructive payloads, and bypass network-based firewalls. Now-infamous crypto-ransomware even encrypts victim's computer documents until a ransom is paid. Therefore, detecting cryptographic functions in binary code is an appealing approach to complement existing malware defense and forensics. However, pervasive control and data obfuscation schemes make cryptographic function identification a challenging work. Existing detection methods are either brittle to work on obfuscated binaries or ad hoc in that they can only identify specific cryptographic functions. In this paper, we propose a novel technique called *bit-precise symbolic loop mapping* to identify cryptographic functions in obfuscated binary code. 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.**

***Keywords*-Cryptographic Function Detection; Obfuscated Binaries; Symbolic Execution.**

I. Introduction

The benefits of cryptographic functions have led to their broad adoption by malicious software (malware) developers. For example, malware developers actively encrypt protocols to bypass network-based firewalls or filters [1], [2], [3]; malicious payloads are often encrypted to impede anti-malware scanning [4], [5]. Recently, crypto-ransomware (e.g., CryptoLocker and CryptoWall) have become an emerging threat that they encrypt infected users' personal files, and victims are forced to pay a ransom to recover their data [6], [7]. Easily accessed cryptographic libraries such as OpenSSL and Microsoft Cryptography API also make reusing cryptographic functions a trivial task [8], [9], [10].

On the other side, to investigate malicious intents and design corresponding defensive solutions, security analysts try to figure out the particular cryptographic functions used in malware binary code [11], [12], [13], [14], [15]. In general, cryptographic function detection facilitates malware analysis and forensics [16] in three ways. First, cryptographic function provides a starting point for analysis. By analyzing or monitoring the execution of the cryptographic functions, security analysts can get access to the plain text and discover the real malicious payloads [17]. Second, cryptographic function identification can save time for analysts to perform binary analysis. If one code section is detected as an implementation of some specific cryptographic algorithm, analysts can skip that section and focus on other parts [18]. Finally, the using of similar cryptographic functions provides valuable clues about malware lineage inference [19], [20]. For example, the same buggy TEA implementation found in both Storm Worm and Silent Banker malware reveals that they are very likely originated from the same authors [11]. However, skilled malware developers can easily apply various code obfuscation techniques to camouflage the telltale signs of cryptographic algorithm implementations [21]. As a result, detecting cryptographic functions in obfuscated binaries has become an important but also challenging work.

A notable difference of cryptographic algorithms from other applications is that they involve a large number of arithmetic computations, which in turn reveals many data related specifications, such as "magic" constant values, excessive use of bitwise operations, stable data flow graphs, and unique input-output relationship. Existing methods for cryptographic function identification in binaries have fully utilized these specific features as detection heuristics. One category is to search and identify static signatures (e.g., instruction chains and mnemonic-const values) inside the binary program [1], [14], [22], [23], [24]. More recent work identifies symmetric cryptographic algorithms by measuring data flow graph isomorphism [15]. Due to the fundamental limitations of static analysis [25], [26], [27], the effects of static detection are severely restricted when analyzing obfuscated binaries. In contrast, dynamic detection captures runtime characteristics [11], [12], [13], [18], [28], [29], which are more resilient to many obfuscation methods. Especially, some advanced detection signatures are only visible at run time, such as the avalanche effect of input-output dependencies [13], [18] and unique input-output relations [11], [12], [29]. However, current dynamic approaches have two major limitations: 1) they are not general enough to detect all commonly used cryptographic functions (e.g., stream or asymmetric ciphers); 2) since many solutions need to recover input and output

parameters from memory, they still suffer from simple data obfuscation schemes (e.g., data encoding [30], [31]).

In this paper, we continue dynamic cryptographic function detection study and present a novel approach, *CryptoHunt*, to address the limitations of existing work. 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 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 [32]. We indeed identified such a non-standard XTEA implementation that reveals a different key schedule constant [33]. 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 novel 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 paper is organized as follows. Section II introduces background and related work. Section III presents an overview of CryptoHunt. Section IV to IX discuss the details of each step in our method. Section X describes our implementation details. We present our evaluation results in Section XI. Discussions and limitations are presented in Section XII. We conclude the paper in Section XIII.

II. Related Work

In this section, we first introduce different code obfuscation techniques that can be used to obfuscate cryptographic function in binaries. These obfuscation schemes are exactly what our study attempts to solve. We then present existing cryptographic function detection work, which can be divided into two categories, static and dynamic methods. The drawbacks of previous work inspire our proposed solution. Next, we introduce literature on symbolic execution and binary difference analysis, which are the most related research work to CryptoHunt.

A. Code Obfuscation

Code obfuscation techniques, which are first designed to protect software intellectual property [34], deliberately transform code to make it more difficult to understand. Nowadays malware authors also heavily rely on code obfuscation to evade detection [21]. One frequently used obfuscation technique in malware is binary packing [5], which first compresses or encrypts an executable binary into data and then recover the original code when the packed program starts running. Since a packing tool typically transforms whole binary code, it may not be suited to obfuscate code snippet such as cryptographic function. In this paper, we focus on defeating another two pervasive obfuscation methods: control obfuscation and data obfuscation. Control obfuscation, such as control flow flattening [35] and opaque predicate [36], greatly changes control flow information to impede reverse engineering. Therefore, cryptographic functions' intra-procedural control flow graphs can be heavily cluttered. Data obfuscation is intended to conceal data value and usage. For example, data encoding schemes [30], [31] convert a variable representation to an obscure one, while data aggregation [37] changes how a variable or array is aggregated. Recovering high-level data abstractions and types from binary code is already pretty hard [38], [39], and data obfuscation will make it more challenging. Since cryptographic algorithms exhibit many specific integer constants and arithmetic computations, data obfuscation becomes particularly fit to hide those attributes of cryptographic function.

B. Static Cryptographic Function Detection

Static crypto detection methods detect cryptographic functions in binaries prior to execution. They perform static analysis to recognize code/data features. Lutz's work [14]

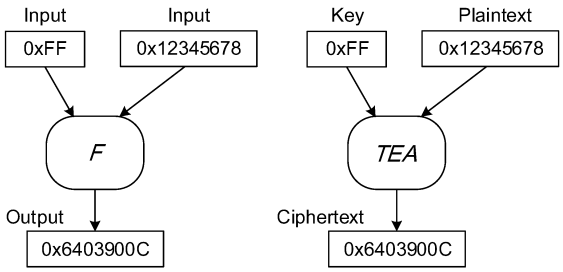


Figure 1: Since F has the same input-output mapping with TEA algorithm, we can recognize F as TEA with an overwhelming probability.

recognizes cryptographic code via three heuristics, such as the presence of loops, entropy, and a high ratio of bitwise operations. Wang et al. [1] utilize a similar method to identify the message decryption phase so as to locate the encrypted data. Matenaar et al. [24] apply multiple detection heuristics such as entropy, constant value, and crypto API. Lestringant et al. [15] 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 [26]. Calvet et al. [11] have demonstrated a very lightweight data obfuscation scheme (splitting a const value into two smaller numbers) can fail static detection.

C. Dynamic Cryptographic Function Detection

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 [13] 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. [12] 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 1). Aligot [11] 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 [11], [12], [29] 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 1); 2) precisely recover I/O parameters from memory (e.g., extract the input and output values); 3) F in Figure 1 must have a perfect match. However, a skilled attacker can easily break down these assumptions. For example, the smallest parameter size the current approaches 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) can complicate parameter extraction. Also, Base64 encoding is commonly found in malware to disguise their malicious payloads [31], 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 [32], [33] so that F in Figure 1 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.

D. Symbolic Execution

Being first proposed by King [40], 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 [41], [42], [43]. EXE [44] automatically detects bugs in C code. KLEE [45] is capable of automatically generating test cases that achieve high path coverage. BAP platform [46], the successor of BitBlaze [47], 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.

E. Binary Difference Analysis

Another related field to our work is automatically finding semantic differences/similarities in binaries [19], [48], [49], [50], [51], [52], [20], [53], which has a wide application in practice, such as malware lineage inference [19], [20], software plagiarism detection [49], [53] and cross-architecture

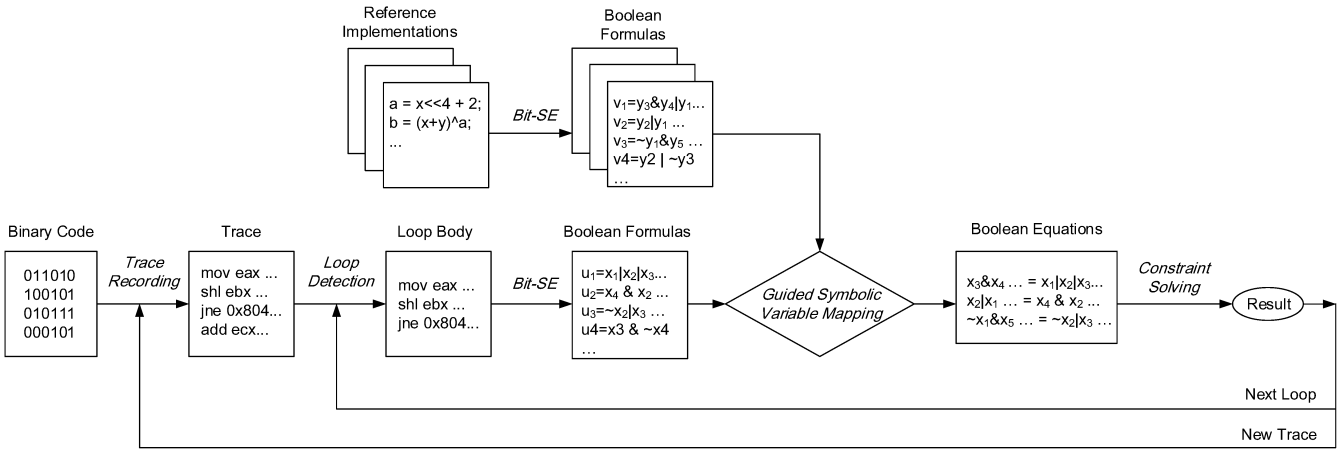


Figure 2: An overview of CryptoHunt's workflow. The words in italics represents CryptoHunt's key components, and "*Bit-SE*" stands for bit-precise symbolic execution.

bug search [51], [52]. 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 [48], [49], [51], [52], [53] and identifying function in stripped binaries [50], which make them not competent to our task. Second, much previous work also compares binaries with symbolic execution and constraint solving [19], [48], [49], [20], [53]. 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.

```
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 3: A reference implementation of TEA.

III. 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 2 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 de-obfuscation [54], [55] and analyzing self-modifying code [56], 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 [11], [12], 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 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.

IV. 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 transformation iterations with CIL [57]. The main loop in Figure 3 shows such a key transformation iterations in TEA cipher. Section XI-A4 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 V and VII. 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 3 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 XI-B.

V. 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 [58]. 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 [59], [56] to renew trace recording when the execution flow returns to the original entry point.

VI. 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 [11] 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 4 shows two trace examples according to the loop definition. In Figure 4(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 4(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 [11] only detects the case in Figure 4(a). Our loop identification method covers both cases in Figure 4.

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 [11]. One extension in our loop identification algorithm is matching function call/return instructions pairs. Function calls could break the loop definition in Figure 4(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

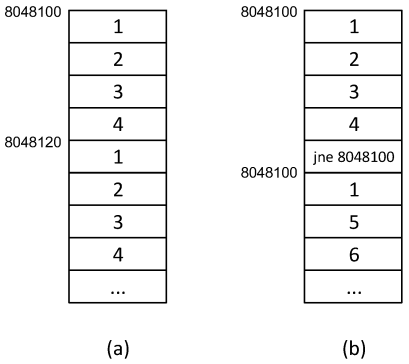


Figure 4: Loop identification in an execution trace.

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 4(b), `[1,2,3,4]` and `[1,5,6]` are two iterations of the same loop. By computing the hash value of each 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 5 shows the folding procedure. In Figure 5(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 5(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 5(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 [2]. 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

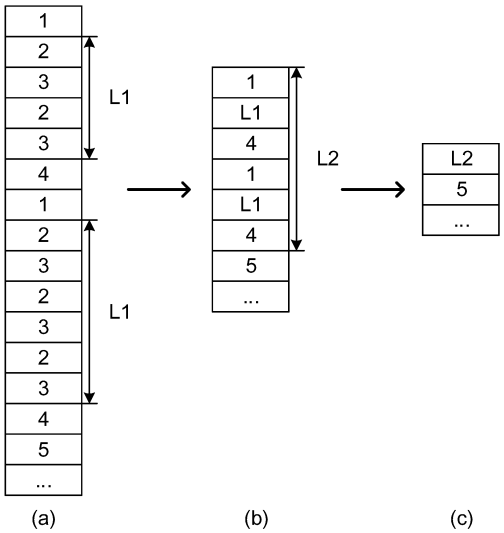


Figure 5: Nested loops identification.

to have a high information entropy [14]. The loop iterations after filtering are passed to the following phases for future analysis.

VII. 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 [11] utilize the input/output relation to identify cryptographic functions. One

```
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: An example of data obfuscation.

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.

Figure 6(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(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(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 [60] and variable merging [37].

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(a) and (b) into boolean formulas, we will find that the two sets of formulas are essentially doing the same task.

VIII. 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.

A. Motivation

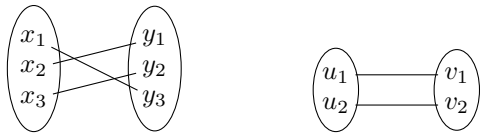
Before describing the detail matching algorithm, first we provide an example to show why we need a mapping algorithm. Suppose we are comparing two loop bodies. One is from the target program and the other one comes from the reference program. The operations in both loop bodies have been translated to two sets of boolean functions as shown in function set (1) and (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.

$$F = \begin{cases} u_1 = f_1(x_1, x_2, x_3) = x_1 \wedge x_2 \vee x_3 \\ u_2 = f_2(x_1, x_2, x_3) = \neg x_1 \vee \neg x_3 \wedge x_2 \end{cases} \quad (1)$$

$$G = \begin{cases} v_1 = g_1(y_1, y_2, y_3) = \neg(y_1 \wedge y_2) \wedge y_3 \\ v_2 = g_2(y_1, y_2, y_3) = y_1 \vee (y_2 \wedge y_3) \end{cases} \quad (2)$$

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 7.

Assuming the mappings in Figure 7 have been found, we can build a boolean equation set (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



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

Figure 7: Variable mapping.

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.

$$\begin{cases} x_1 \wedge x_2 \vee x_3 = \neg(x_2 \wedge x_3) \wedge x_1 \\ \neg x_1 \vee \neg x_3 \wedge x_2 = x_2 \vee (x_3 \wedge x_1) \end{cases} \tag{3}$$

B. 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 combination of setting only one input variable to 1 and the remainder to 0. An input identity matrix enumerates all possible input combinations following this rule.

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. Therefore, the mapping problem is essentially equivalent to the following problem:

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

This is the key idea in our mapping algorithm. Notice that swapping 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 $M_{n,m}^O$ 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 each row in $M_{n,m}^O$. Essentially it describes the fact that how many outputs is evaluated to 1 when setting a specific input to 1 and leave the remainder to 0. Similarly, a column sum vector describes how many times each output variable is set to 1 in $M_{n,m}^O$. $\vec{rv}$ and $\vec{cv}$ are used to compute the mapping in a given output matrix.

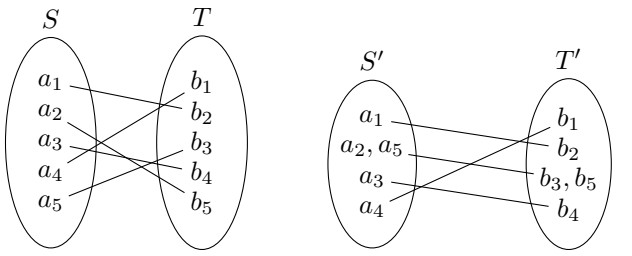
For example, given an output matrix $M_{2,3}^O$, its $\vec{rv}$ and $\vec{cv}$ are shown as follows.

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

So far we have defined all concepts related to the formulas in this paper. 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 kinds of mapping, *full mapping* and *partial mapping*. As shown in Figure 8(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 8(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 $a_2 \mapsto b_5, a_5 \mapsto b_3$.

C. Algorithm Description

So far we have introduced the basic concepts that are needed to formalize the algorithm. As mentioned above, we



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

Figure 8: Mapping examples.

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 only swapping rows and columns.

Based on this idea, we propose the variable mapping algorithm which is described in Algorithm 1. Briefly speaking, our mapping algorithm is seeking for all possible mappings given a series of specific inputs, which are generated based on already mapped inputs. The method is complete, which means if a mapping exists, it must appear in the result. It is probable 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 can be viewed 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 consistency.
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.

Algorithm 1 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

D. Example

In the last section, we have described the formal definition of the mapping algorithm. Now we provide an example to show the whole procedure. We still use F and G as shown in formula (1) and (2). We initiate the partial mapping list L set as follows. M_{in} and M_{out} are initiated as empty.

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

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 (4) and the matrix in (5).

$$\begin{cases} \{x_1 = 1, x_2 = 0, x_3 = 0\} \Rightarrow \{u_1 = 0, u_2 = 0\} \\ \{x_1 = 0, x_2 = 1, x_3 = 0\} \Rightarrow \{u_1 = 0, u_2 = 1\} \\ \{x_1 = 0, x_2 = 0, x_3 = 1\} \Rightarrow \{u_1 = 1, u_2 = 0\} \end{cases} \tag{4}$$

$$M_1^I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, M_1^O = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{5}$$

Following the same method, we feed the inputs into G and the result is shown in (6) and (7).

$$\begin{cases} \{y_1 = 1, y_2 = 0, y_3 = 0\} \Rightarrow \{v_1 = 1, v_2 = 0\} \\ \{y_1 = 0, y_2 = 1, y_3 = 0\} \Rightarrow \{v_1 = 0, v_2 = 0\} \\ \{y_1 = 0, y_2 = 0, y_3 = 1\} \Rightarrow \{v_1 = 0, v_2 = 1\} \end{cases} \quad (6)$$

$$M_2^I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, M_2^O = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \quad (7)$$

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.

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

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

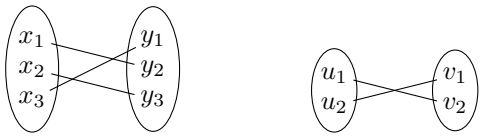
Similarly, we create the row and column sum vectors for M_1^O and M_2^O.

$$r\vec{v}_1 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, c\vec{v}_1 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, r\vec{v}_2 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, c\vec{v}_2 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

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

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

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 9.



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

Figure 9: Final result of variable mapping.

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

$$\begin{cases} x_1 \wedge x_2 \vee x_3 = x_3 \vee (x_1 \wedge x_2) \\ \neg x_1 \vee \neg x_3 \wedge x_2 = \neg(x_3 \wedge x_1) \wedge x_2 \end{cases} \quad (8)$$

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.

IX. 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.

Table I: Cryptographic algorithm categories.

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

X. 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 [58] (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 [46] (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 [61] as the theorem prover. For the convenience of future research, we have released CryptoHunt source code at https://github.com/s3team/CryptoHunt.

XI. 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.

A. Answer to RQ1: Crypto Libraries

1) Dataset: We first test CryptoHunt with commonly used cryptographic algorithms from four categories (see Table I). We choose TEA and AES as block cipher examples. Tiny Encryption Algorithm (TEA) [62] is a simple block cipher, which is frequently adopted by malware authors to hide malicious intent; while the Advanced Encryption Standard (AES) [63] 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 [64] as the hashing algorithm since it is widely used on the Internet for software integrity checking. At last, we use RSA [65] 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 I, 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

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 [66] is an assembly tool that identifies cryptographic programs by static signatures. Similarly, both Findcrypto2 [67] and Signsrch [23] are IDA [68] plug-in tools and search static signatures for cryptographic function detection. DFGIsom [15] statically identify symmetric cryptographic algorithms and their parameters inside binary code based on Data Flow Graph (DFG) isomorphism[4]. Kerchkhoffs [12] 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

[1] https://www.openssl.org/

[2] https://www.gnu.org/software/libgcrypt/

[3] https://github.com

[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 [69].

parameters. The advanced detection tool, Aligot [11], relies on identifying unique input-output relations at loop boundary.

3) Obfuscation Options: To obfuscate cryptographic algorithm implementations, we rely on a state-of-the-art compile-time obfuscation tool, Obfuscator-LLVM [70], which supports popular obfuscation transformations [34], [71]. 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*.

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 [62] 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* [72].

4) Evaluation Result: The evaluation result is shown in Table II[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.

a) TEA: TEA is a 64-bit cipher which uses 128-bit key. It is usually implemented as 64 rounds of Feistel structure [62]. In CryptoHunt, we use the transformations inside one Feistel structure loop as the reference implementation (see Figure 3). As shown in Table II, 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 version, only CryptoHunt is able to detect the highly obfuscated TEA algorithm. In another implementation TEA*, the result is the same.

Table II: 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							✓

b) AES: The AES design is based on substitution-permutation network [63], 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

[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.

box without checking the details inside, it cannot perform more fine-grained detection as CryptoHunt.

c) 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 II 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.

d) MD5: MD5 algorithm [64] 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.

e) RSA: The RSA cryptographic algorithm [65] 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 II 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 4(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.

```
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 10: 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 11: The decryption function in an Apache Module injection malware.

B. 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 10 shows a reference implementation of the decryption procedure in XTEA. However, malware authors have already abused such flexibility to produce new variations to evade detection. A recent study [33] 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 11 presents the core part of the new XTEA in C code.

In Figure 11, 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 11 as a C program. The source code shown in Figure 10 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 III.

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

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

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

Table IV: Evaluation result on malware samples.

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

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.

C. Answer to RQ1: Malware Samples

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

From Table IV, 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 XI-A4: 1) big-integer encoding; 2) incomplete loop identification.

D. 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 V, our dataset includes GNU 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.

Table V: False positive evaluation dataset.

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

Table VI: 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

E. 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 [58] to record the execution trace. The online trace logging overhead is typically 5-6X slowdown. Table VI 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 VII: 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

F. 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 II. We collect the number of identified loops, number of mapping variables candidates, and number of STP queries. The result is shown in Table VII. 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.

XII. Discussion

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 [41], [73]. The static analysis may consider multiple paths. However, the various obfuscation methods adopted by malware authors will undoubtedly impede the accurate static analysis [26]. 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 [74] 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 [75], [76], [77]. 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 [78]).

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.

XIII. Conclusion

Nowadays cryptographic functions have been widely adopted by malware developers to disguise their payloads, escape from network analysis, and in general, hide their malicious behaviors. Detecting cryptographic functions in binary code can help security analysts to figure out malicious intents and design defensive solutions. However, due to the prevalence of code obfuscation techniques, cryptographic function detection has become a very challenging work. Existing detection methods are far from mature. Their effects are restricted when handling obfuscated binary code. In this paper, we propose a new technique called *bit-precise symbolic loop mapping* to first capture the specific features of cryptographic algorithms with boolean formulas, which are later used as signatures to efficiently match 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.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) grants CNS-1223710 and CCF-1320605, and the Office of Naval Research (ONR) grants N00014-13-1-0175, N00014-16-1-2265, and N00014-16-1-2912. Ming was also supported by the University of Texas System Rising STARs Program.

References

[1] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "Reformat: Automatic reverse engineering of encrypted messages," in *Proceedings of the 14th European Conference on Research in Computer Security (ESORICS'09)*, 2009.

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

[3] J. Calvet, C. R. Davis, and P.-M. Bureau, "Malware authors don't learn, and that's good!" in *Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE'09)*, 2009.

[4] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *Proceedings of the 36th IEEE Symposium on Security & Privacy*, 2015.

[5] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Computing Surveys*, vol. 46, no. 1, 2013.

[6] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda, "Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks," in *Proceedings of the 12th International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'15)*, 2015.

[7] Bromium Labs, "Understanding Crypto-Ransomware," http://www.bromium.com/sites/default/files/bromium-report-ransomware.pdf.

[8] F. Leder and T. Werner, "Know your enemy: Containing conficker," The Honeynet Project, Tech. Rep., 2009.

[9] P. Porras, H. Saidi, and V. Yegneswaran, "Conficker C P2P Protocol and Implementation, September 2009."

[10] G. Tenebro, "Waledac–an overview," https://www.symantec.com/connect/blogs/waledac-overview, 2009, Symantec Official Blog.

[11] J. Calvet, J. M. Fernandez, and J.-Y. Marion, "Aligot: Cryptographic function identification in obfuscated binary programs," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, 2012.

[12] F. Gröbert, C. Willems, and T. Holz, "Automated identification of cryptographic primitives in binary programs," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID'11)*, 2011.

[13] X. Li, X. Wang, and W. Chang, "CipherXRay: Exposing cryptographic operations and transient secrets from monitored binary execution," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 2, March 2014.

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

[15] P. Lestringant, F. Guihéry, and P.-A. Fouque, "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)*, 2015.

[16] C. H. Malin, E. Casey, and J. M. Aquilina, *Malware Forensics: Investigating and Analyzing Malicious Code*. Syngress, 2008.

[17] W. Yan, Z. Zhang, and N. Ansari, "Revealing packed malware," *IEEE Security & Privacy*, vol. 6, no. 5, pp. 65–69, 2008.

[18] J. Caballero, P. Poosankam, S. McCamant, D. Babi ć, and D. Song, "Input generation via decomposition and re-stitching: Finding bugs in malware," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[19] J. Ming, D. Xu, and D. 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'15)*, 2015.

[20] ——, "MalwareHunt: semantics-based malware diffing speedup by normalized basic block memoization," *Journal of Computer Virology and Hacking Techniques*, 2016.

[21] P. OKane, S. Sezer, and K. McLaughlin, "Obfuscation: The hidden malware," *IEEE Security and Privacy*, vol. 9, no. 5, 2011.

[22] I. Levin, "Draft Crypto Analyzer (DRACA)," http://www.literatecode.com/draca.

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

[24] F. Matenaar, A. Wichmann, F. Leder, and E. Gerhards-Padilla, "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)*, 2012.

[25] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*, 2003.

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

[27] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," in *Proceedings of the 16th USENIX Security Symposium (USENIX Security'07)*, 2007.

[28] D. D. Hosfelt, "Automated detection and classification of cryptographic algorithms in binary programs through machine learning," Master's thesis, Johns Hopkins University, March 2015.

[29] R. Zhao, D. Gu, J. Li, and R. Yu, "Detection and analysis of cryptographic data inside software," in *Proceedings of the 14th International Conference on Information Security (ISC'11)*, 2011.

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

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

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

[33] J. Grunzweig, "Digging into the new apache injection module," https://www.trustwave.com/Resources/SpiderLabs-Blog/Digging-Into-the-New-Apache-Injection-Module/, 2013, SpiderLabs Blog.

[34] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," The University of Auckland, Tech. Rep., 1997.

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

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

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

[38] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS'10)*, 2010.

[39] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," in *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[40] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

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

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

[43] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proceedings of the International Conference on Software Engineering (ICSE'13)*, 2013.

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

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

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

[47] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *4th International Conference on Information Systems Security. Keynote invited paper*, 2008.

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

[49] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "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)*, 2014.

[50] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *23rd USENIX Security Symposium (USENIX Security'14)*, 2014.

[51] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[52] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and T. H. B. Kuan, "BinGo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'16)*, 2016.

[53] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, 2017.

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

[55] J. Ming, D. Xu, L. Wang, and D. Wu, "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)*, 2015.

[56] L. Martignoni, M. Christodorescu, and S. Jha, "OmniUnpack: Fast, generic, and safe unpacking of malware," in *Proceedings of the 23rd Annual Computer Security Applications Conference(ACSAC'07)*, 2007.

[57] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*, 2002.

[58] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "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)*, 2005.

[59] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the hidden-code extraction of unpack-executing malware," in *Proceedings of the 2006 Annual Computer Security Applications Conference (ACSAC'06)*, 2006.

[60] W. Zhu, C. Thomborson, and F.-Y. Wang, "Applications of homomorphic functions to software obfuscation," in *Proceedings of the 2006 International Workshop on Intelligence and Security Informatics (WISI'06)*, 2006.

[61] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the International Conference on Computer Aided Verification (CAV'07)*, 2007.

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

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

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

[65] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[66] Chun, "x3chun's cryptosearcher," http://x3chun.reteam.org/, 2004.

[67] I. Guilfanov, "Ida-pro/plugins/findcrypt2," https://www.aldeid.com/wiki/IDA-Pro/plugins/FindCrypt2, 2015, aldeid.

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

[69] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[70] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, 2015.

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

[72] D. Williams, "The tiny encryption algorithm (tea)," *Network Security*, pp. 1–14, 2008.

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

[74] Q. Zhao, I. Cutcutache, and W.-F. Wong, "PiPA: Pipelined profiling and analysis on multicore systems," *ACM Transactions on Architecture and Code Optimization*, vol. 7, no. 3, Dec. 2010.

[75] D. Kirat and G. Vigna, "MalGene: Automatic extraction of malware analysis evasion signature," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[76] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, "Detecting environment-sensitive malware," in *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID 2011), Menlo Park, CA, USA*, September 2011.

[77] D. Kirat, G. Vigna, and C. Kruegel, "BareCloud: Bare-metal analysis-based evasive malware detection," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[78] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008.