BinGold: Towards Robust Binary Analysis by Extracting the Semantics of Binary Code as Semantic Flow Graphs (SFGs)[☆]

Saed Alrabaee, Lingyu Wang, Mourad Debbabi

Computer Security Laboratory, Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada

Abstract

Binary analysis is useful in many practical applications, such as the detection of malware or vulnerable software components. However, our survey of the literature shows that most existing binary analysis tools and frameworks rely on assumptions about specific compilers and compilation settings. It is well known that techniques such as refactoring and light obfuscation can significantly alter the structure of code, even for simple programs. Applying such techniques or changing the compiler and compilation settings can significantly affect the accuracy of available binary analysis tools, which severely limits their practicability, especially when applied to malware. To address these issues, we propose a novel technique that extracts the semantics of binary code in terms of both data and control flow. Our technique allows more robust binary analysis because the extracted semantics of the binary code is generally immune from light obfuscation, refactoring, and varying the compilers or compilation settings. Specifically, we apply data-flow analysis to extract the semantic flow of the registers as well as the semantic components of the control flow graph, which are then synthesized into a novel representation called the semantic flow graph (SFG). Subsequently, various properties, such as reflexive, symmetric, antisymmetric, and transitive relations, are extracted from the SFG and applied to binary analysis. We implement our system in a tool called *BinGold* and evaluate it against thirty binary code applications. Our evaluation shows that *BinGold* successfully determines the similarity between binaries, yielding results that are highly robust against light obfuscation and refactoring. In addition, we demonstrate the application of *BinGold* to two important binary analysis tasks: binary code authorship attribution, and the detection of clone components across program executables. The promising results suggest that *BinGold* can be used to enhance existing techniques, making them more robust and practical.

1. Introduction

Reverse-engineering has become a crucial process in many important applications of digital forensics, ranging from program analysis to malware detection. Their importance stems from the fact that there are many practical situations in which the original high-level source code is unavailable, has been lost, or is otherwise inconvenient to use. The objective of reverse-engineering often involves understanding the control and data-flow structures of the functions in the given binary code [10]. However, this is usually a challenging task, as binary code inherently lacks structures as a result of using jumps and symbolic addresses, highly optimized control flow, varying registers, memory location based on the processor and compiler, and the possibility of interruptions [9].

To this end, several studies have been conducted on binary analysis in various domains. Certain existing techniques are designed to identify clone functions [26, 35], reused functions [8, 41], standard compiler libraries recognition [23, 39], authorship attribution [7, 40], and function recognition [2, 11, 42]. The features used in such techniques can be categorized into three main groups: syntactic (e.g., n-gram [26]), semantic (e.g., longest common subsequence of semantically equivalent basic blocks [30]), and structural features (e.g., control flow graph [8]). However, the majority of these features may rely critically on assumptions concerning specific compilers and compilation settings. To make things worse, it is well known that techniques such as refactoring and light obfuscation (Section 6.4) can significantly alter the structure of the code, even for simple programs. Applying such techniques or changing the compiler and compilation settings may thus lead to variability in the precision/recall of many existing techniques [31].

Problem statement. In this paper, we focus on the challenging problem of enabling binary analysis techniques to resolve the aforementioned issues using their existing features. We leverage both data-flow and control-flow analysis in order to extract the semantics of the code, which is then integrated into a novel representation called a Semantic Flow Graph (SFG). The extracted semantics

[☆]This research is the result of a fruitful collaboration between the Computer Security Laboratory (CSL) of Concordia University, Defence Research and Development Canada (DRDC) Valcartier and Google under a DND/NSERC Research Partnership Program.

of the code may enable existing binary analysis techniques to withstand light obfuscation or refactoring, since the extracted semantics of the binary code is generally immune to light obfuscation, refactoring, and varying compilers or compilation settings. In addition, there are several other benefits in extracting the semantics of the code automatically. For instance, the semantics could represent the fingerprint of a code. In this regard, binary fingerprinting approaches generally suffer from a common problem: typically, fingerprints are overly sensitive to even the slightest of changes to the binary code. The capability of extracting the semantics of the code, which is more robust than syntax-based fingerprints, would help immensely in finding the right selection of fingerprints or in modifying existing fingerprints in a reliable manner. For example, while the machine learning approach for identifying an author in [40] may suffer from the compiler's effects [7], the features could lead to more robust results if we apply them based on the semantics of the code, as we will show in section 6.6.

Our approach. We begin by applying different levels of normalization to the assembly code. The normalization process allows the user to generalize the assembly instructions (i.e., memory references, registers, and constant values) to different levels. Subsequently, we color each basic block according to the proposed categories (to be discussed in section 4.2). When a color is added to each basic block, we apply data-flow analysis to capture the internal structure of a function, and we also leverage the data flow among basic blocks to capture data and variable dependencies. This representation of data and control dependencies is considered a reliable representation; among instructions, these data and control dependencies remain unchanged regardless of the order of instructions [Qiu et al.]. The semantics of a control flow graph (CFG) is represented by constructing a conservative approximation of target function prototypes by means of use-def analysis of possible callees. We then couple these results with liveness analysis at each indirect callsite to arrive at a many-to-many relationship between callsites and target callees to recover callsite and callee signatures. Finally, we integrate both types of semantics into a novel representation called a Semantic Flow Graph (SFG). Various properties, such as reflexivity, symmetry, and transitivity relations, are then computed over the SFG as inputs to binary analysis tasks.

Results overview. We design and implement a prototype that applies our techniques to facilitate data and control analysis. In the current prototype, we focus on tackling the effects of compilers, compilation settings, light obfuscation techniques, and refactoring tools, which resolves many limitations of existing works. We evaluate our tool by applying it to a set of projects. We also evaluate the time efficiency of our tool. To justify our claim of improving the accuracy of existing works, we re-evaluate certain existing tools after applying our techniques and show that the obtained accuracy is superior to the previously reported accuracy.

Contributions. In summary, our main contributions are the following:

- We enumerate the effects that compilers, compilation settings, light obfuscation modes, or refactoring tools have on binaries and propose a solution for dealing with such effects in binary analysis. Our approach is fully automatic and does not require a priori knowledge concerning the source of the compiler. The novel way in which we model these effects also makes our approach amenable to faster machine-learning algorithms.
- We introduce various techniques to recover different semantic components of binary code from both data and control flow analyses of code. We also combine these extracted semantic components into a novel representation called a Semantic Flow Graph.
- We test our method on a large test suite across different operating systems, compilers, and compiling optimizations. Our results show that our method achieves higher accuracy than previously available fingerprint representations.
- We re-evaluate certain existing binary analysis techniques after integrating them with our system, and superior obtained accuracy.

Roadmap. We first provide background information in Section 2. We introduce a motivating example in Section 3. Section 4 describes the *BinGold* system in detail. Our detection system is introduced in Section 5. Experimental results, followed by some discussions, are presented in Section 6. Limitations and a conclusion are presented in Section 7.

2. Background

As previously mentioned, most existing works, especially for fingerprinting applications (e.g., authorship), do not tolerate certain binary disturbances such as compiler optimizations or differences in build environments. In what follows, we will describe examples of such disturbances.

2.1. Function inlining

In practice, the compiler may inline a small function into its caller code as an optimization. This may introduce additional complexity to the code. Furthermore, function-inlining significantly changes the CFG of a program, which may become problematic for existing binary analysis approaches. Finding inlined code is a challenging task [Qiu et al.]. The accuracy will undoubtedly drop if the features are derived from a function that includes inlined functions or if the target programs do not show such inlining. Still, using the multiple initial basic block matches will not likely find the multiple counterparts in the non-inlined target program. We thus use data-flow analysis, which provides the ability to find code that has been inlined.

2.2. Instruction Reordering

Compilers may reorder independent computations to enhance data locality. Reordered instructions in a basic block change the syntactic representation. However, the semantics of a basic block remain the same. By normalizing the code, our system performs reordering at no additional cost.

2.3. Common Subexpression

Common subexpression elimination is a classical compiler optimization technique used to remove redundant computations. If two identical operations share the same set of input operands, they clearly produce the same output. In fact, in x86 code, effective address computation is generally performed every time a memory access is made [28]. As a consequence, it is difficult to detect whether two memory accesses are made at the same location since their address operands systematically belong to different instructions.

2.4. Constant Folding

Constant folding is the process of recognizing and evaluating constant expressions at compile-time rather than computing them at runtime. Terms in constant expressions are typically simple literals, such as the integer literal 2, but may also be variables whose values are known at compile-time. The process is also related to compiler optimizations used by many modern compilers. An advanced form of constant propagation known as “sparse conditional constant propagation” can more accurately propagate constants and simultaneously remove dead code (for instance, 2 + 4 becoming 6 at compile-time)[21]. Either way, using sampling, the output variables will have equal values. This process has the additional advantage of comparing semantics instead of syntax.

2.5. Calling Conventions

The method of transferring parameters to a function is not always well-standardized. In many cases, it is possible to specify a particular calling convention in a C++ declaration; for example, `_cdecl`, `_stdcall`. This specifies which registers are used for transferring parameters. For instance, `ecx`, `edx` means that the first parameter goes into `ecx`, the second parameter goes into `edx`, and subsequent parameters are stored on the stack. Furthermore, the calling of statement functions is implementation-defined; due to the fact that they are only locally defined, the compiler has the freedom to apply any calling convention that is deemed appropriate [15]. Our system handles these effects by abstracting from concrete register names. It is not important which registers (or stack/memory addresses) are used to pass registers or to return results. Having different registers for passing different values would be problematic when comparing the syntax of the instruction representations. To this end, some optimizations modify the CFG (e.g., loop unrolling, dead code elimination) and may become problematic.

2.6. Refactoring Process

Refactoring is the process of changing the structure of code without changing the way it behaves [19]. Refactoring is considered a best practice when creating and maintaining software; indeed, research suggests that programmers practice it regularly [34, 44]. Examples of refactoring include renaming a variable, moving a method from a superclass to its subclasses, and taking a few statements and extracting them into a new method. These examples are referred to as `RENAME, PUSH DOWN METHOD, and EXTRACT METHOD` [19].

3. Motivating Example

We start with a simple example composed of part of MD5 written in C++ (Listing 1). In this sample, the hex representation of the digest is returned as a string. MD5 performs many binary operations on the “message” (text or binary data) to compute a 128-bit “hash”. We compile this part of the MD5 example code on Windows 7 using g++, Visual Studio 2010, XCODE, and ICC. We then use IDA to disassemble the binary. Many security tools use IDA in this way, as a first step before performing additional analysis [22, 36].

```
std::string MD5::hexdigest() const {
    if (!finalized)
        return "";

    char buf[33];
    for (int i=0; i<16; i++)
        sprintf(buf+i*2, "%02x", digest[i]);
    buf[32]=0;

    return std::string(buf);
}
```

Listing 1: Motivating example: Part of MD5 method

We compute the control flow graph for the fragment and then compare them as illustrated in Table 2. We notice through the motivating example that the compiler also makes changes to both the control structure and the basic blocks and hence instructions. We show a list of traditional features in Table 1.

We name the graphs as graph A, graph B, graph C, and Graph D; these graphs represent CFGs from visual studio, ICC, g++, and XCODE, respectively. We can see in Table 2 that among some graphs, there are features with the same values; for example the number of nodes is the same for graphs A and B. Cyclomatic complexity varies; it is calculated by $M = E - N + 2P$, where E is the number of edges, N is the number of nodes, and P is the number of connected components. Additionally, we

Table 1: Graph Features Description

Feature	Description
Number of nodes	Number of basic blocks
Number of edges	Number of control flows (i.e., true)
K-cone	K represents the number of CFG level
Radius	Minimum vertex eccentricity
Width of graph	Maximum number of nodes at the same level
Length of graph	Number of nodes in the longest path
Diameter	The longest shortest path between any two nodes in the graph
Cyclometry Complexity	Number of linearly independent paths within the CFG

observe there are some common values between graphs A and C. For instance, the number of nodes is 8 when it is compiled with Visual Studio, but it is 13 with g++ and 5 with XCODE. Additionally, the number of edges ranges from 4 to 17.

Table 2: Graph features applied on CFGs for the fragment code in Listing 1, which is compiled by visual studio, ICC, g++, and XCODE

Feature	Graph A	Graph B	Graph C	Graph D
# of nodes	8	8	13	5
# of edges	9	8	15	4
K-cone	0-4	0-6	0-4	0-3
Radius	2	3	5	2
Width of graph	3	2	4	2
Length of graph	5	7	5	4
Diameter	3	4	6	2
Cyclometry Complexity	3	2	4	1

As a result of the aforementioned differences, the structural approaches may lead to false positives by claiming that two graphs are the same (because of similar graph features), when in fact they are not. Additionally, we observe through the motivating example that there are differences in instructions at the syntax level; these differences affect the results of the syntax approaches in terms of reporting similarities. Hence, the necessity of having an automated tool that can simply extract the semantics of a code will significantly reduce the percentage of false positives.

4. Extracting the Semantics of Binary Code

In this section, we describe how we built upon the background in Section 2 to perform the task of extracting the semantics of a binary code.

4.1. Architecture overview

Our architecture employs a series of techniques illustrated in Figure 1 and described in the upcoming sections. First, the binary code is disassembled by IDA Pro[2] disassembler. Second, a set of rules are applied to assembly instructions to normalize the code. Third, data flow rules are applied to these normalized instructions to construct data flow dependencies. In addition, we extract the semantics of the CFG by constructing a conservative approximation of the target function prototype by means of a use-def analysis of possible callees. We then couple these results with liveness analysis at each indirect call site to arrive at a many-to-many relationship between call sites and target callees in order to recover call site and callee signatures. Both types of semantics are integrated into a new representation called the Semantic Flow Graph (SFG). Subsequently, the properties of the SFG, such as the reflexive, symmetric, anti-symmetric, and transitive relations are extracted from the SFG.

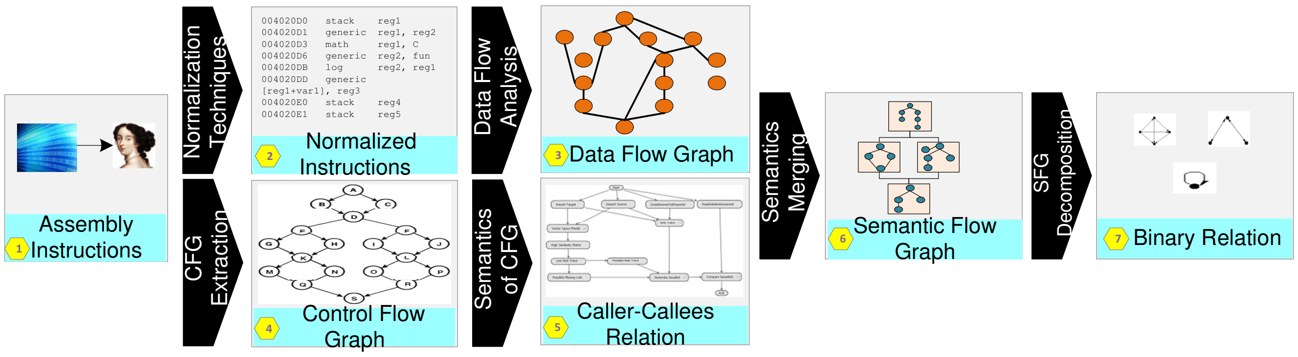
4.2. Normalization

The first step is to disassemble the input binaries into a collection of assembly files. To do so, using a disassembler such as IDA Pro [2] is the common way. Each assembly file contains a set of functions. Each function contains a sequence of assembly code instructions, and each assembly instruction consists of a mnemonic and a sequence of operands. Mnemonics are used to represent the low-level machine operations. The operands can be classified into three categories: memory reference, register reference, and constant value. We may have two fragments of a code that are identical both structurally and syntactically, but differ in terms of memory references [17]. For example, two instructions with the same mnemonic but with different registers, such as `eax` or `ebx`, can be considered identical. Thus, it is essential that the assembly code be normalized prior to comparison. The objective of the normalization step is to generalize the memory references, registers, and constant values to an appropriate, user-selected level. For constant values, the normalizer generalizes them to VAL, which simply ignores the exact constant value. The same logic applies to memory references. For registers, the user can generalize them according to the various levels of normalization. The top-most level REG generalizes all registers, regardless of type. The next level differentiates General Registers (e.g., eax, ebx), Segment Registers (e.g., cs, ds), and Index and Pointer Registers (e.g., esi, edi). The third level breaks down the General Registers into three groups by size–namely, 32, 16, and 8-bit registers.

Table 3: Groups of instruction code

Group	**Code**	**Example**
Stack	s	push
Arithmetic	a	add
Logical	l	xor
Compare	c	test
External call	e	call ds:scoket
Internal call	i	call sub:xxx
Conditional jump	cj	jle
Unconditional jump	uj	jmp
Generic	g	move

After normalizing the instructions as described above, we convert each instruction to a three-tuple $IT = (g, c, d)$, where g represents the group to which that instruction belongs, c represents the characteristics of the opcode, and



```
004020D0   stack    reg1
004020D1   generic  reg1, reg2
004020D3   math     reg1, C
004020D6   generic  reg2, fun
004020DB   log      reg2, reg1
004020DD   generic
[reg1+var1], reg3
004020E0   stack    reg4
004020E1   stack    reg5
```

Figure 1: Architecture Overview

d represents the instruction opcode. The digits in *g* denote the type that the opcode belongs to according to Table 3. The digits in *c* denote the types of operands. For example, c = 1 for "op reg", c = 11 for "op reg, reg", and c = 12 for "op reg, mem". All characteristic values for an instruction with 0, 1, or 2 operands are listed in Table 4.

Table 4: The code for different operand types

	Operand 2			
Operand 1	**No**	**Register**	**Memory**	**Constant**
No	0	1	2	3
Register	1	11	12	13
Memory	2	21	22	23
Constant	3	31	32	33

Finally, *IT = (g,c,d)* is converted to a number called the ID of the instruction by invoking the function ID(IT). The function *ID(IT) = Hash(g)|(c)|(d)* is applied to obtain a 32-bit integer, which is considered the ID of the instruction. Hash() is a string hash function that maps different instruction mnemonics to different 16-bit integers.

4.3. Data Flow Graph Construction

After normalizing the instructions, we apply data flow to infer the program variable relations using coarse reasoning about the program control flow and data dependencies. Depending on how such analyses choose to model the flow of information through the data structures. Let R_k/W_k denote registers or memory that instruction I_k reads or writes. If i_1 and i_2 are instructions belonging to I and they are in the same basic block, then we define the following possible dependencies: i_1 writes something which will be read by i_2; i_1 reads something before i_2 overwrites it; and i_1 and i_2 both write the same variable. This category of dependency is considered an internal dependency. The other dependency is control dependence. If i_1 and i_2 are both in the same basic block, and i_2 is a control instruction, we call it an internal control dependence. Also, i_1 and i_2 are in two different basic blocks, where i_1 is the last instruction in the first basic block and i_2 is executed in the second basic block as the first instruction, where the second basic block is a successor of the first basic block in the control flow graph, then it is also an internal control dependence.

4.4. Semantics of a CFG

Intuitively, we try to construct the semantics of the control flow graph by deploying a combination of two type-based control flow invariants: target-oriented invariants and callsite-oriented invariants. Target-oriented invariants are based on traditional approaches [13], and callsite-oriented invariants have been explored recently for binaries [45]. We believe that this invariant, inspired by source-based CFG techniques [43], is applied at the binary level for the first time. We demonstrate that even a binary-only approach with this concept can be both efficient and effective. As noted, extracting complete function and call-site type information at the binary level is difficult in practice. Therefore, a relaxed form of type information, function argument count, and many-to-many type-based matching strategy is realized between callsites and targets.

To compute arguments for each callsite, our system performs a conservative backward static analysis. We explain this analysis as follows. We use static analysis to determine the argument count at the callee side, and given a set of address-taken (AT) functions, our system iterates over each function and performs custom inter-procedural constant propagation analysis [25]. The analysis focuses on collecting state information on registers, to determine whether or not they are used for passing arguments. A register can exist in one of the following states: read-before-write (R), write-before-read (W), or clear/untouched (C). The total state of a particular basic block contains the combined information for all possible argument registers. The analysis starts at the entry basic block of an AT function and iterates over the instructions to determine the usage of registers. If all argument registers are either R or W, the analysis terminates. However, if at least one register is in a C state, a recursive forward-edge analysis starts, and continues until the block has no outgoing edges. We assume that the target writes all arguments and stop the recursion, thus transforming all remaining clear registers into a W state. Forward static analysis for a basic block *B* that has n outgoing edges provides us with a set of states $S_i (i = 1, 2, \cdots, n)$. These states represent argument usage information for each path following edge i. Each state is represented by a vector comprising the status of each one of the six argument registers.

We define the following invariants:

Definition 1. An indirect callsite cs is said to be type Ω_{max} (max $= 0, 1, 2, \cdots$) if it prepares at most max function arguments (referred to as actuals).

Definition 2. A function f has its address taken if and only if the address of f is loaded into memory/registers.

4.5. *Equivalence Relations and Partitions in SFG*

The data flow graph together with the invariants form the semantic flow graph. We combine those semantic information to form a new representation in order to facilitate more efficient graph matching between different binary codes for determining the similarity or integrating into some existing frameworks. Formally, a semantic flow graph (SFG) is defined as follows.

Definition 3. A semantic flow graph $G = (N, V, \zeta, \gamma, \vartheta, \lambda, \omega)$ is a directed attributed graph where N is a set of nodes, $V \subseteq (N \times N)$ is a set of edges and ζ is edge labeling function which assigns a label to each edge: $\zeta \longrightarrow \gamma$, where γ is a set of labels (internal dependency or external dependency). ϑ is a call-callee relation function which colors each node $n \in N$ based on its relation with other node $k \in N$. Finally, ω is a function for coloring dataflow control or data dependencies.

We illustrate a simple example in Figure 2 to show how SFG could be constructed. As shown, ω is a function for coloring dataflow dependencies; control or data dependency. ϑ is a call-callee relation function. We can notice the green color in Figure 2 (c) represents caller-callee relation. For instance, *i2* has a caller-callee relation with *i5*. Besides,

We then construct the relations from the SFG. We generalize equivalence relations and equivalence classes, where an equivalence relation on a set of features (semantics features) F is a relation $R \subset F \ x \ F$ such that:

- $(f_i, f_j) \ \varepsilon \ R$ for all $f \varepsilon F$, which is called the reflexive property
- $(f_i, f_j) \ \varepsilon \ R$ implies $(f_j, f_i) \ \varepsilon \ R$
- (f_i, f_j) and $(f_j, f_k) \ \varepsilon \ R$ imply (f_i, f_k)

We also extract a collection of nonempty sets of features F, which is called partition P. This is a collection of nonempty sets $f_1, f_2, ...$ such that $f_i \bigcap f_j = for \ i \neq j$ and $\bigcup_k F_k = X$. Let $\sim$ be an equivalence relation on a set F and let $f \varepsilon F$. Then $[f] = \{f_j \varepsilon F : f_j \sim f_i$ is called the equivalence class of $f\}$.

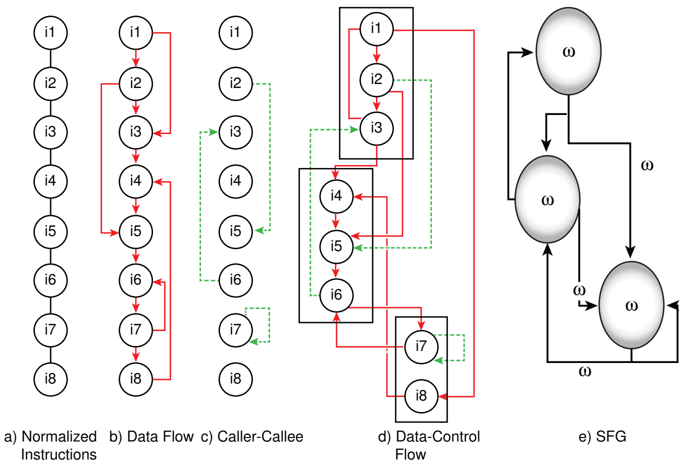


Figure 2: An example of constructing SFG

5. Detection Process

We next describe the detection system *Bingold*. Since *Bingold* extracts different types of features that capture the semantics of code, the detection system is composed of multiple components employing a series of techniques, as depicted in Figure 3 and explained in the next subsections.

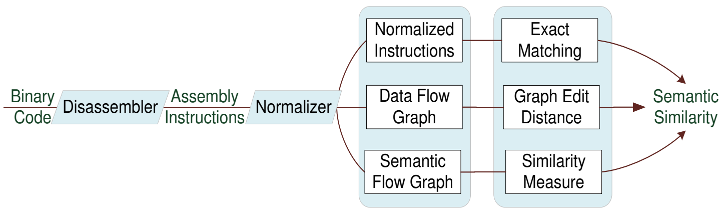


Figure 3: Detection System

5.1. *Exact Matching*

As previously described, we normalize the code according to predefined rules and then apply the predefined categories to those normalized instructions. We then convert those instructions to hash vectors. Finally, we match instructions together.

5.2. *Graph Edit Distance*

For inexact matching between data flow graphs, a distance metric is needed. In this paper, we employ the graph edit distance for this purpose. The edit distance between two graphs measures their similarity in terms of the number of edits required to transform one into the other. Given two data flow graphs, to transform one graph into another, we define two concepts: `internal flow dependency` and `external flow dependency`. The edit distance between two data flow graphs G and H is thus defined as the minimum weight of all dependencies d between them; i.e.,

$sim(G,H) = min\ w(V_{G,H})$, where V is the function for checking the dependencies.

We define the dissimilarity between two data flow graphs G and H as follows:

Definition 4. The dissimilarity $\rho(G,H)$ between two graphs is a value ranging between $[0,1]$, where 0 indicates the highest similarity between graphs and 1 indicates the lowest similarity, as calculated by:

$$\rho(G,H) = \frac{w(V_{G,H})}{|N_G| + |N_H| + |V_G| + |V_H| + |\varrho_G| + |\varrho_H|}$$

where $w(V_{G,H})$ is the weighted cost of dependencies, $|N_G|$ and $|N_H|$ are the number of data flow nodes, $|V_G|$ and $|V_H|$ are the number of internal dependencies, and $|\varrho_G|$ and $|\varrho_H|$ are the number of external dependencies in G and H, respectively.

5.3. Similarity Measure

For the extracted relations, we compare two graphs in terms of the similarity of their reflexive, symmetric, and transitive relations. Given two data SFGs G and H, we define the similarity measure $sim(G,H) = max\ R(V_{G,H})$. R is a function extracts the common relations between two graphs and measures the similarity between them.

5.4. Weight Parameter Settings

We define for each component (data flow, caller-callee relationship, and SFG) in our system a weight. These wights are: α, β, and γ, to determine the contribution of each component. We experimentally determine the optimal values for these parameters. The parameter setting is computed using nine-fold cross-validation. We evaluate values of α ranging from 0 to 1 in steps of size 0.1 and β ranging from 0 to 1 in steps of size 0.1. For a given choice of α, β, and γ, it is required that $\alpha + \beta + \gamma = 1$. In each setting, the features are extracted using our system and the F_1 score is computed that the maximum F_1 score is obtained for $\alpha = 0.5$, $\beta = 0.2$, and $\gamma = 0.3$. We use these values as the default for *BinGold* as well as throughout the rest of the evaluation.

6. Evaluation

This section details the evaluation of our system. Section 6.1 describes the experimental setup. Section 6.2 presents the evaluation metrics. Section 6.3 shows the results of our system for different compilers and compilation settings. Section 6.4 shows the robustness of our system against light obfuscation techniques and the refactoring process. Finally, section 6.6 shows the effect of integrating our system into certain existing approaches and demonstrates improvements in accuracy.

6.1. Experimental Setup

Our experimental setup comprises our dataset, the ground truth used, and the selection of system weight parameters.

6.1.1. Dataset

We evaluate our system against 30 programs for which we have the source code. These programs are only used to extract the ground truth by compiling the source code with debugging information.

Table 5 summarizes the 30 programs. For each program, the table shows the program identifier, the program name, the binary code statistics, and the source compiler. From the binary code it captures the type of executable generated (PE or ELF) and the number of functions in the executable. The binary code information is extracted using IDA pro [2] by reading the executable's debugging information. 3 projects compiled by 4 compilers, 8 projects compiled by 3 compilers, and 19 projects compiled by 2 compilers. The dependency of the program restricts us to compiling each project using 4 compilers.

Table 5: Programs used in our system evaluation

		Binary Code		
ID	Program	Type	Funct	Compiler
1	SQlite	PE	3920	VS, GCC, ICC, XCODE
2	OpenSSL	PE	2163	VS, GCC
3	info-zip	PE	1784	VS, ICC
4	jabber	PE	5910	VS, GCC
5	Hashdeep	PE	2905	VS, XCODE, GCC
6	libpng	PE	9226	VS, GCC
7	ultraVNC	PE	3526	VS, GCC
8	lcms	PE	1082	XCODE, ICC, GCC
9	ibavcodec	PE	739	VS, GCC, ICC
10	TrueCrypt	PE	1093	VS, GCC
11	libjsoncpp	PE	4114	VS, ICC
12	7z	PE	2179	VS, GCC, ICC
13	7zG	PE	2530	VS, GCC, ICC
14	7zFM	PE	3149	VS, GCC, ICC
15	lzip	ELF	33	VS, GCC
16	tinyXMLTest	ELF	2744	VS, GCC, ICC, XCODE
17	libxml2	ELF	58	VS, GCC, ICC
18	Mersenne Twister	ELF	2740	VS, GCC
19	bzip2	ELF	285	VS, GCC
20	lshw	ELF	1429	VS, GCC
21	smartctl	ELF	457	VS, GCC
22	pdftohtml	ELF	499	VS, GCC, XCODE
23	ELF statifier	ELF	2340	VS, GCC
24	FileZilla	PE	6250	VS, GCC
25	ncat	PE	1855	VS, GCC
26	Hasher	PE	436	VS, GCC, ICC, XCODE
27	tfshark	ELF	439	VS, GCC
28	dumpcap	ELF	448	VS, GCC
29	tshark	ELF	1008	VS, GCC
30	pageant	ELF	2212	VS, GCC

Our dataset are open-source projects from SourceForge [5], and the GNU software repository [4]. Our dataset includes 17 PE binaries and 13 ELF binaries. We include multiple programs from the same project that could be compiled by different compilers and use those programs to analyze the applicability and efficiency of our system.

6.2. Evaluation Metrics

To evaluate the accuracy of our system, we conducted the following experiments. First, we compared two sets of results: the results output by some existing tools (i.e., authorship attribution, clone detection) and the results after integrating our system with these tools. Second, we compared the similarity of the same program when it is compiled by different compilers and with different compilation settings. Third, we applied different light obfuscation techniques to the same binary file and checked the similarity based on the semantic information extracted by our system. Finally, we applied different refactoring techniques to the source code and compiled it using different compilers. We then employed our tool to measure the similarity between the binary files.

We use validity metrics such as *precision*, *recall*, and F_1. Precision *(P)* and recall *(R)* are defined as follows:

$$P = \frac{TP}{TP + FP}, \; R = \frac{TP}{TP + FN} \tag{1}$$

where *TP* (true positives) is the number of functions assigned correctly by our system; *FP* (false positives) is the number of functions assigned incorrectly by our system; and *FN* (false negatives) is the number of functions not assigned by our system but which actually belong to it. To combine both precision and recall, we use the F_δ score with $\delta = 1$, which is equal to the harmonic mean of the precision and recall values. F_1 scores fall within the interval $[0, 1]$, where the larger the F_1 score, the better the overall accuracy. Since our application domain is much more sensitive to false positives than to false negatives, we use the F-measure as follows.

$$F_1 = 2 \cdot \frac{PR}{P + R} \tag{2}$$

6.3. Accuracy Results of C/C++ Programs with Different Compilers and Compilation Settings

As previously mentioned, we compiled 30 programs using different compilers such as XCODE and ICC. We evaluate how well our system detects the similarities among those executables using the F_1 score. Table 6 summarizes the results. The median F_1 score is 0.78. The precision ranges from 0.60 to 0.90, and the recall ranges from 0.64 to 0.92.

The accuracy of the C++ results is higher than the accuracy of the C results because C++ source code contains classes with small-sized methods. These small components are mostly unaffected by compilers or compilation settings. However, they may be inlined and are thus easily identified based on data flow components. For instance, the program FileZilla has the highest F_1 score of 0.90, while the program dumpcap has the lowest F_1 score of 0.63. For C programs, the median F_1 score is 0.67. The results for C binary code similarity are worse than the results for C++ programs. This is expected as C programmers are not constrained by the object-oriented paradigm and often place functions with different semantics in the same source file. For example, the file tfshark.c in tfshark combines string processing, message processing (read/write/print), and common functions for program output. These functions are technically similar in semantic representation, but the presence of all three reduces the F_1 score to 0.64 when using automated ground truth based on source files. Moreover, C programs have less modularity than C++ programs so it may be harder to extract the semantics of a code.

6.4. Accuracy Results after Applying Light Obfuscation and Refactoring Techniques

We consider a random set of 15 files from our dataset and compile them using Visual Studio 2010. The binaries are converted into assembly files through the disassembler, and the code is then obfuscated using the `DaLin` generator [29]. This generator applies the following light obfuscation: (i) register renaming (RR), which is one of the oldest and simplest techniques used in metamorphic generators; (ii) Instruction reordering (IR), which transposes instructions that do not depend on the output of previous instructions; (iii) Dead code insertion (DCI), which injects a piece of code that has no effect on program execution (i.e., may not execute or may execute with no effect); and (iv) equivalent instruction replacement (EIR). We perform initial tests on the selected files and report the accuracy measurements. Light obfuscation is then applied and new accuracy measurements are obtained and observed.

We used existing open-source tools for the C++ refactoring process [3, ref]. We consider the techniques of i) Renaming a variable (RV), ii) Moving a method from a superclass to its subclasses (MM), and iii) Extracting a few statements and placing them into a new method (NM). In-depth explanations of these techniques are detailed in [19].

The results are shown in Table 7. The results shown in the table demonstrate that our system performs well in identifying similarities; however, we obtain lower accuracy when we apply refactoring as opposed to when we apply light obfuscation.

6.5. Time Efficiency

The running time for extracting the semantics of code is measured by considering the total time spent during each step: normalization process, extracting the semantics of the data flow, extracting the semantics of the control flow, and forming the SFG by extracting the binary relations. In the semantic extraction process, the binary application is first disassembled using IDA pro [2], and features are then extracted by running our IDApython script. The assembly instructions must first be normalized and hashed to a unique value. This process of extracting the features takes 15 seconds for the smallest application in our dataset

Table 6: Our system accuracy in determining the similarity between binaries

Program	Precision	Recall	F1	Program	Precision	Recall	F1
SQlite	0.75	0.88	0.81	tinyXMLTest	072	0.79	0.75
OpenSSL	0.72	0.66	0.69	libxml2	0.78	0.82	0.80
info-zip	0.68	0.9	0.77	Mersenne Twister	0.78	0.88	0.83
jabber	0.67	0.88	0.76	bzip2	0.82	0.9	0.86
Hashdeep	0.63	0.72	0.67	lshw	0.83	0.83	0.83
libpng	0.82	0.68	0.74	smartctl	0.89	0.92	0.90
ultraVNC	0.81	0.67	0.73	pdftohtml	0.85	0.75	0.80
lcms	0.75	0.66	0.70	ELF statifier	0.83	0.74	0.78
ibavcodec	0.77	0.81	0.79	FileZilla	0.90	0.92	0.90
TrueCrypt	0.90	0.88	0.89	ncat	0.72	0.71	0.71
libjsoncpp	0.85	0.67	0.75	Hasher	0.71	0.68	0.69
7z	0.74	0.77	0.73	tfshark	0.70	0.65	0.67
7zG	0.66	0.81	0.73	dumpcap	0.62	0.64	0.63
7zFM	0.66	0.82	0.76	tshark	0.60	0.68	0.64
lzip	0.66	0.9	0.75	pageant	0.67	0.67	0.67

(which is dumpcap) and 45 seconds for the largest application (libpng) on a Windows 32-bit machine with 16GB RAM. Extracting the first part of the semantics (data flow) takes 20 seconds for dumpcap and 60 seconds for libpng, while extracting the second part of the semantics (control flow) takes 23 seconds for dumpcap and 26 seconds for libpng. The last step, forming the new representation and extracting the relations described in Section 4.5, takes 10 seconds for dumpcap and 14 seconds for libpng. Based on those results, we believe our system will be efficient enough for most real world applications.

Table 7: Results after applying light obfuscation techniques and the refactoring process

Method	Precision	Recall	F1
RR	0.89	0.88	0.88
IR	0.91	0.92	0.91
DCI	0.87	0.93	0.90
EIR	0.81	0.82	0.81
RV	0.87	0.90	0.88
MM	0.85	0.82	0.83
NM	0.67	0.72	0.70

6.6. Applications

In this section, we demonstrate the applicability of our system to two applications: authorship attribution and clone detection.

Previous work has demonstrated that it is possible to identify the authors of binary code [7, 40]. However, existing approaches usually assume that the compiler and its settings are known. In addition, the features used in such techniques are sensitive to any light obfuscation or refactoring. Hence, we apply our system to the binary and then re-examine their features based on the outputs of our system. Regarding clone detection, some existing works have demonstrated the use of K-CFG [26], Tracelet, n-grams [26, 40], idioms [26, 40], RFG [7], and strings [26]. Both authorship and clone accuracy are greatly improved by integrating our tool with the aforementioned tools, as shown in Table 8.

Dataset. The dataset we use for authorship attribution originates from Google Code Jam 2010 [6]. It consists of single-authored programs. For each author, there are multiple programs as the Code Jam is a multi-round programming contest. The dataset therefore provides a perfect benchmark for authorship attribution, and data from Google Code Jam has been used in all recent program authorship studies (e.g., [7], [40]). Regarding clone detection, we use 10 programs from our dataset (1-10).

Evaluation. Because those applications domain is much more sensitive to false positives than false negatives, we use the F-measure as follows.

$$F_{0.5} = 1.25 \cdot \frac{PR}{0.25P + R} \qquad (3)$$

Because each component in our system can handle one or more effects, our system could enhance the application of existing works. For instance, the normalization can handle compiler effects, data flow analysis can identify inline functions, the caller-callee relationship can tackle the refactoring process, and the relation extracted from the SFG can handle most light obfuscation. Results are summarized in Table 8.

According to the results in Table 8, we can conclude that our tool leads to substantial improvements in the accuracy of existing work. For instance, it improves the accuracy of clone systems (e.g., idioms) by 16%, which is a considerable improvement. Another example considers the Tracelet system, since it already includes normalization techniques and data flow analysis, our tools only provide the benefit of semantics in terms of control flow graph,

Table 8: The effect of integrating BinGold to certain existing works

Feature	$F_{0.5}$ **(Before applying BinGold)**	$F_{0.5}$ **(After applying BinGold)**	**Application**
Idioms [40]	0.71	0.80	Authorship
Idioms [26]	0.72	0.88	Clone
Graphlet [40]	0.60	0.76	Authorship
RFG [7]	0.72	0.79	Authorship
Call graphlet [40]	0.64	0.71	Authorship
K-CFG [26]	0.78	0.877	Clone
Tracelet [16]	0.66	0.70	Function Fingerprinting

which leads to 4% improvement of accuracy.

7. Related Work

Extracting the semantics of a binary code has attracted a great deal of research in different areas. In what follows, we briefly review previous work in these areas.

7.1. Frameworks for Extracting Semantics of Binary Code

There are several frameworks proposed for extracting the semantics of binary code for particular tasks, such as BinSlayer [12], BinJuice [27], BitShred [24], and iBinHunt [32]. BinSlayer uses a polynomial algorithm to find the similarity between executables, obtained by fusing the well-known BinDiff algorithm [18] with the Hungarian algorithm [33] for bi-partite graph matching. BinJuice extracts the abstraction of the semantics of binary blocks which is termed "juice". Whereas the denotational semantics summarizes the computation performed by a block, its juice presents a template of the relationships established by the block. BitShred is a framework for automatic code reuse detection in binary code [24]. BitShred can be used for identifying the amount of shared code based on the ability to calculate the similarities among binary code. iBinhunt is a technique to find the semantic differences between two binary programs when the source code is not available. It uses the process of analyzing control flow, particularly intra-procedural control flow [32].

7.2. Binary Code Characterization

Several approaches have used semantic or behavioral patterns to characterize binary code in the anti-malware community [14, 20]. These approaches identify patterns in the externally visible behavior of programs, such as interactions with the operating system (through system calls or standard libraries) or manipulation of the file system; for example, Fredrikson et al. form malware specifications based on the sequence of system calls and their arguments observed at runtime [20]. Recently, in [38], the authors propose a novel approach to characterize a binary by identify library functions. They introduce execution dependence graphs (EDGs) to describe the behavior characteristics of binary code. Then, by finding similar EDG subgraphs in target functions, they identify both full and inline library functions. These works cannot be directly applied to improve the accuracy of existing works because, unlike our system which extracts semantics as generic inputs to other works, those tools are designed for specific tasks.

8. Limitations, Future Work, and Concluding Remarks

Our work has a few important limitations. First, the system is unlikely to achieve accurate results if the authors packed their binary or used advanced obfuscation techniques. Second, although we have tested our work on four popular compilers, the effectiveness against other compilers remains to be evaluated. Third, we have not investigated the impact of different platforms such as ARM, MIPS, etc. in this study.

We point out four main avenues for future research in terms of improving our system. First, we suggest researching the applicability of our method to other compiler. Second, we suggest investigating more platforms such as ARM. Third, we seek to extend our system to include visualizations of the semantics of binary code rather than presenting numeric results. Finally, while we have demonstrated the viability of our system to enhance existing works in a variety of applications, a more thorough investigation of different applications is necessary.

To conclude, we have designed a system called *BinGold* for accurately and automatically recovering the semantics of a binary code. Our experimental results indicate that the approach is efficient in terms of computational resources and could thus be considered a practical approach to real-world binary analysis. Moreover, the experimental results suggest that *BinGold* can be used to enhance existing techniques, making them more robust and practical.

Acknowledgments

The authors thank the anonymous reviewers for their valuable comments. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

References

[ref] Refactoring tool. `https://www.devexpress.com/Products/CodeRush/`. Accessed on Feb, 2016.

[2] (2011). HexRays: IDA Pro. `https://www.hex-rays.com/products/ida/index.shtml`. Accessed on Feb, 2016.

[3] (2016). C++ refactoring tools for visual studio. `http://www.wholetomato.com/`. Accessed on Feb, 2016.

[4] (2016). Gnu software repository. `www.gnu.org/software/software.html`. Accessed on Feb, 2016.

[5] (2016). Sourceforge. `http://sourceforge.net`. Accessed on Feb, 2016.

[6] (2016). The Google Code Jam. Available from:. `http://code.google.com/codejam/`. Accessed on Feb, 2016.

[7] Alrabaee, S., Saleem, N., Preda, S., Wang, L., and Debbabi, M. (2014). Oba2: an onion approach to binary code authorship attribution. *Digital Investigation*, 11:S94–S103.

[8] Alrabaee, S., Shirani, P., Wang, L., and Debbabi, M. (2015). Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12:S61–S71.

[9] Balakrishnan, G. and Reps, T. (2010). Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):23.

[10] Balliu, M., Dam, M., and Guanciale, R. (2014). Automating information flow analysis of low level code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1080–1091. ACM.

[11] Bao, T., Burket, J., Woo, M., Turner, R., and Brumley, D. (2014). Byteweight: Learning to recognize functions in binary code. In *USENIX Security Symposium*.

[12] Bourquin, M., King, A., and Robbins, E. (2013). Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 4. ACM.

[13] Budiu, M., Erlingsson, U., and Ligatti, J. (2005). Control-flow integrity. Citeseer.

[14] Christodorescu, M., Jha, S., and Kruegel, C. (2008). Mining specifications of malicious behavior. In *Proceedings of the 1st India software engineering conference*, pages 5–14. ACM.

[15] Christodorescu, M., Kidd, N., and Goh, W.-H. (2005). String analysis for x86 binaries. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 88–95. ACM.

[16] David, Y. and Yahav, E. (2014). Tracelet-based code search in executables. In *ACM SIGPLAN Notices*, volume 49, pages 349–360. ACM.

[17] Farhadi, M. R., Fung, B. C., Fung, Y. B., Charland, P., Preda, S., and Debbabi, M. (2015). Scalable code clone search for malware analysis. *Digital Investigation*, 15:46–60.

[18] Flake, H. (2002). Graph-based binary analysis. *Blackhat Briefings 2002*.

[19] Fowler, M. (1999). *Refactoring: improving the design of existing code*. Pearson Education India.

[20] Fredrikson, M., Jha, S., Christodorescu, M., Sailer, R., and Yan, X. (2010). Synthesizing near-optimal malware specifications from suspicious behaviors. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 45–60. IEEE.

[21] Glesner, S. and Blech, J. O. (2004). Classifying and formally verifying integer constant folding. *Electronic Notes in Theoretical Computer Science*, 82(2):410–425.

[22] Hu, X., Chiueh, T.-c., and Shin, K. G. (2009). Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620. ACM.

[23] Jacobson, E. R., Rosenblum, N., and Miller, B. P. (2011). Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 1–8. ACM.

[24] Jang, J. and Brumley, D. (2009). Bitshred: Fast, scalable code reuse detection in binary code (cmu-cylab-10-006). *CyLab*, page 28.

[25] Khedker, U., Sanyal, A., and Sathe, B. (2009). *Data flow analysis: theory and practice*. CRC Press.

[26] Khoo, W. M., Mycroft, A., and Anderson, R. (2013). Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 329–338. IEEE Press.

[27] Lakhotia, A., Preda, M. D., and Giacobazzi, R. (2013). Fast location of similar code fragments using semantic'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 5. ACM.

[28] Lestringant, P., Guihéry, F., and Fouque, P.-A. (2015). Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 203–214. ACM.

[29] Lin, D. and Stamp, M. (2011). Hunting for undetectable metamorphic viruses. *Journal in computer virology*, 7(3):201–214.

[30] Luo, L., Ming, J., Wu, D., Liu, P., and Zhu, S. (2014). Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400. ACM.

[31] Meng, X. and Miller, B. P. (2015). Binary code is not easy.

[32] Ming, J., Pan, M., and Gao, D. (2013). ibinhunt: Binary hunting with inter-procedural control flow. In *Information Security and Cryptology–ICISC 2012*, pages 92–109. Springer.

[33] Munkres, J. (1957). Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38.

[34] Murphy-Hill, E., Parnin, C., and Black, A. P. (2012). How we refactor, and how we know it. *Software Engineering, IEEE Transactions on*, 38(1):5–18.

[35] Myles, G. and Collberg, C. (2005). K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 314–318. ACM.

[36] Pappas, V., Polychronakis, M., and Keromytis, A. D. (2012). Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 601–615. IEEE.

[Qiu et al.] Qiu, J., Su, X., and Ma, P. Using reduced execution flow graph to identify library functions in binary code.

[38] Qiu, J., Su, X., and Ma, P. (2015). Library functions identification in binary code by using graph isomorphism testings. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 261–270. IEEE.

[39] Rahimian, A., Shirani, P., Alrbaee, S., Wang, L., and Debbabi, M. (2015). Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation*, 14:S146–S155.

[40] Rosenblum, N., Zhu, X., and Miller, B. P. (2011). Who wrote this code? identifying the authors of program binaries. In *Computer Security–ESORICS 2011*, pages 172–189. Springer.

[41] Ruttenberg, B., Miles, C., Kellogg, L., Notani, V., Howard, M., LeDoux, C., Lakhotia, A., and Pfeffer, A. (2014). Identifying shared software components to support malware forensics. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–40. Springer.

[42] Shin, E. C. R., Song, D., and Moazzezi, R. (2015). Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 611–626.

[43] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., and Erlingsson, . (2014). Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*.

[44] Xing, Z. and Stroulia, E. (2006). Refactoring practice: How it is and how it should be supported-an eclipse case study. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 458–468. IEEE.

[45] Zhang, M. and Sekar, R. (2013). Control flow integrity for cots binaries. In *Usenix Security*, pages 337–352.