Detecting Fine-Grained Similarity in Binaries

By

ANDREAS SÆBJØRNSEN

B.S. Physics (University of Oslo, Norway) 2003

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Zhendong Su, Chair

Premkumar Devanbu

Kent Wilken

Daniel Quinlan

Committee in Charge

2014

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Contents

Andreas Sæbjørnsen
July 2014
Computer Science

Detecting Fine-Grained Similarity in Binaries

Abstract

Large software projects contain significant code duplication, mainly due to copying and pasting code. Many techniques have been developed to identify similar code to enable applications such as refactoring, detecting bugs, and detecting illicit code reuse. Because source code is often unavailable, especially for third-party software, finding similar code in binaries becomes particularly important. Unfortunately, binaries present challenges that make it difficult to detect similar code, such as overcoming the lack of high-level information and piercing the veil of compiler optimizations. Due to these difficulties, we are yet to have practical techniques and tools for binary similarity detection.

This dissertation presents novel research that tackles these challenges toward realizing practical binary similarity detection. The central focus of the work is on automatically extracting syntactic and semantic signatures directly from the binaries. This dissertation presents a family of related algorithms, frameworks and tools for extracting these signatures. The two main tools presented are a binary clone detection tool for syntactic similarity and SLEUTH, a tool that detects software license violations using hybrid semantic and syntactic signatures. Although software license violations are common, they often remain undetected due to challenges inherent in binary similarity detection. Examples of software license violations include code a programmer wrote for an ex-employer or open source software licensed under copy-left (such as the Linux kernel reused in a closed source router). Each presented algorithm or tool is practical, automatically finding fine-grained similarity with high precision.

This dissertation also introduces a general, mixed binary interpretation framework and its accompanying implementation for realizing the aforementioned work.

Acknowledgments

First and foremost, I would like to thank my family and Tacara Soones. Their dedication, encouragement and love made this work possbile. I would also like to thank my friends and coworkers for their support.

Dr. Dan Quinlan from Lawrence Livermore National Laboratory has been my boss and mentor since I entered the USA for the first time in 2003. He gave me the opportunity to work with excellent colleauges, like Chunhua Liao, Jeremiah Willcock, Markus Schordan, Robb Matzke, and Thomas Panas. Without his support and encouragement, I would not be where I am today.

The engineering challenges we had to be overcome to complete this thesis research were monumental and, at times, seemed insurmountable. Amongst all my colleagues at Lawrence Livermore National Laboratory, I would like to single out Robb Matzke. Robb and I have worked closely together for nearly 4 years now and I have come to appreciate both our professional collaboration and friendship. I am proud of what we have created.

My close collaborator, mentor, sparring partner and friend, Lecturer Earl Barr at University College of London, has worked tirelessly with me on SLEUTH and SCOUT. His support, hard work, encouragement and advice have helped me grow both as a professional and individual.

And finally, I would like to thank Zhendong Su, my patient, brilliant and inspiring doctoral advisor at UC Davis. His clarity and advice have helped me crystalize this thesis research. He has taught me how to define a good research problem and tell its story. Thank you for your patience and support when mining for new knowledge.

CHAPTER 1

Introduction

The convergence of ubiquitous personal computing and powerful software development frameworks have led to an explosion of developed software. Developers commonly reuse code to save time, money and effort. Most proprietary code ships in binary form and so they may contain similar code, hidden behind the binary veil. Although we suspect that a piece of code has been reused, we have no effective mechanism to detect code reuse in binaries.

The discovery of code reuse in binaries — *binary similarity detection* — is the focus of this dissertation, which presents novel algorithms, frameworks and tools for automatically detecting code similarity in binaries.

Two lines of research into clone detection exist, syntactic or semantic. Chapter 2 presents our work on syntactic similarity detection, and Chapter 3 presents our hybrid semantic and syntactic similarity detection technique. Chapter 4 describes a general framework we designed and realized to enable the above work.

Chapter 2: Syntactic Similarity Detection

In most cases, code reuse is simply a common practice. However, it hinders maintenance because each copy is maintained separately. For example, when a bug or security defect is discovered in one copy then all copies should be fixed even if they are incorporated in software maintained by separate groups of developers. Syntactic clone detection techniques use patterns of language structures and repeated sequences of code to find clones. Detecting syntactic source-level clones is a well-explored research area, and many scalable and precise tools exist to solve the problem. However, source code is frequently unavailable such as in the case of commercial off the shelf (COTS) software. In these situations, one needs binary similarity detection to find copying. Yet, we do not have effective tools for detecting similar binary code. To overcome this challenge, we introduce the first practical syntactic clone detection tool for binaries.

Our syntactic clone detection tool is highly scalable in detecting fine-grained code similarity. Our algorithm follows a general tree similarity framework [46]. Instead of performing a quadratic number of pair-wise comparisons of instruction sequences, it models the essential structural information of the instruction sequences with numerical vectors and groups close vectors to identify clones.

In addition to being practically effective, our binary clone detection tool presents novel techniques to generate precise and robust vectors for binaries and to compactly represent the vectors for improved scalability.

Chapter 3: Hybrid Syntactic and Semantic Similarity Detection

Software license violations are common. A license violation occurs when an unscrupulous developer commits code theft by knowingly copying source code from a victim's code base for use in another code base in violation of the victim's license. For example, a closed-source program that includes GPL-ed code violates the GPL. Most proprietary code ships in binary form, thus may contain similar code hidden behind the binary veil. Unfortunately for victims of code theft, binary analysis presents challenges that make it easy to hide illicit code reuse in binaries. Example challenges are the lack of type and other high-level information and code transformations performed by compiler optimizations. Efforts like `gpl-violations.org` have revealed illicit reuse, but the state-of-the-art binary similarity detection is still largely manual because of the inherent difficulties. Manual analysis is not practical because it is time consuming and requires costly specialized labor. Although we may suspect code theft occurs, we have no effective means of detecting theft.

Compiler optimizations alone usually defeat purely syntactic techniques to detect code theft in binaries, *e.g.* the binary clone detection algorithm presented in Chapter 2. However, the key reason for code theft is to reuse the stolen code's behavior, so semantic similarity is unaffected by such transformations. Inspired by the Schwartz-Zippel lemma, we measure semantic similarity by comparing function output on random input. Because semantic similarity detection samples an input domain, it is susceptible to false positives due to innocent, cleanroom implementations of the same specification or the failure of semantic similarity to find an input that triggers divergent behavior. Here, syntactic similarity can filter and increase the precision of semantic similarity. In other words, while behavioral similarity over-approximates actual similarity, syntactic similarity under-approximates it. As each approach alone cannot solve the code theft problem, we develop a hybrid semantic and syntactic algorithm for detecting binary code theft and realized a tool called SLEUTH. This allows it to search for software theft without contending with the combinatorial explosion of binary variants compilers and compiler flags can generate from a single source file. SLEUTH does not require source code.

We have implemented and evaluated our technique on popular open-source programs containing known duplicated code. Results show that our analysis is scalable (analyzed our corpus in a few hours) and precise (produced few false positives). We show that our tool is practical — it detects code theft with over

90% precision and it is resilient to source code obfuscations as well as different compilers and compiler optimizations.

Chapter 4: Mixed Interpretation

One of the biggest challenges of implementing the hybrid syntactic and semantic similarity detection algorithm is creating a semantic signature by exercising a function with randomly generated inputs and extracting the side-effects as output. Exercising a whole program with a random input is well-supported in major binary analysis tools, but it lacks tool support for exercising a function or an arbitrary code fragment with a randomly generated input. Current binary analysis tools either analyze a decompiled binary without implementing a representation of the execution model of the target machine or execute a trace of the program concretely. However, finding a trace that covers a code fragment is an difficult problem. In addition, the requirement of analyzing a trace of a program forces the user to think in terms of the whole program instead of reasoning about a specific code segment. Whole program analysis makes it hard to exercise a function on a random input. It also makes it extremely challenging to separate the side-effects of a function from the rest of the program. To tackle these challenges, we design and implement a framework to interpret binaries at an arbitrary offset.

Our framework, SCOUT, has the unique capability of executing a program concretely or abstractly, starting from any offset. SCOUT can analyze program fragments or functions without a trace that covers the function. The challenge inherent to starting from any offset is handling the initial lack of state. To overcome this challenge, SCOUT accurately models each non-floating-point x86 instruction and how it operates on the heap, stack, and register file. This model and SCOUT 's support for system calls guarantee that no external, unanalyzable functions exist to limit or disrupt SCOUT 's analysis. SCOUT supports symbolic execution [51] and execution over any user-defined abstract domain [28].

Abstract interpretation is a powerful framework for static program analysis. SCOUT is unique in its support of the abstract interpretation of binaries. An abstract interpreter evaluates a program in an abstract domain; a well-chosen abstract domain is critical to obtain a precise and tractable analysis.

Tool Dissemination All the tools presented in this dissertation are open-source and have been integrated with the ROSE compiler infrastructure. Interested readers can download the tools, scripts and data needed to replicate our results from www.rosecompiler.org.

CHAPTER 2

Detecting Code Clones in Binary Executables

Large software projects contain significant code duplication, mainly due to copying and pasting code. Many techniques have been developed to identify duplicated code to enable applications such as refactoring, detecting bugs, and protecting intellectual property. Because source code is often unavailable, especially for third-party software, finding duplicated code in binaries becomes particularly important. However, existing techniques operate primarily on source code, and no effective tool exists for binaries.

In this chapter, we describe the first practical clone detection algorithm for binary executables. Our algorithm extends an existing tree similarity framework based on clustering of characteristic vectors of labeled trees with novel techniques to normalize assembly instructions and to accurately and compactly model their structural information. We have implemented our technique and evaluated it on Windows XP system binaries totaling over 50 million assembly instructions. Results show that it is both scalable and precise: it analyzed Windows XP system binaries in a few hours and produced few false positives. We believe our technique is a practical, enabling technology for many applications dealing with binary code.

2.1. Introduction

Code duplication is common and hinders software maintenance, program comprehension, and software quality. *Clone detection*, the problem of identifying duplicated code, is thus an important problem and has been extensively studied. Many clone detection algorithms exist [6, 33, 46, 48, 52, 56], ranging from basic string-based algorithms [6] to more sophisticated algorithms based on program dependency graphs (PDGs) [33, 52].

Most existing clone detection algorithms operate only on source code, but not on binaries. However, the ability to detect binary clones is important because source code is not always available, for example, in the case of commercial off the shelf (COTS) software. One important application of a practical clone detection algorithm for binaries is the discovery of copyright infringements. A closed-source program could, for example, include GPLed source code in violation of its license; binary-level clone detection may be used to find that.

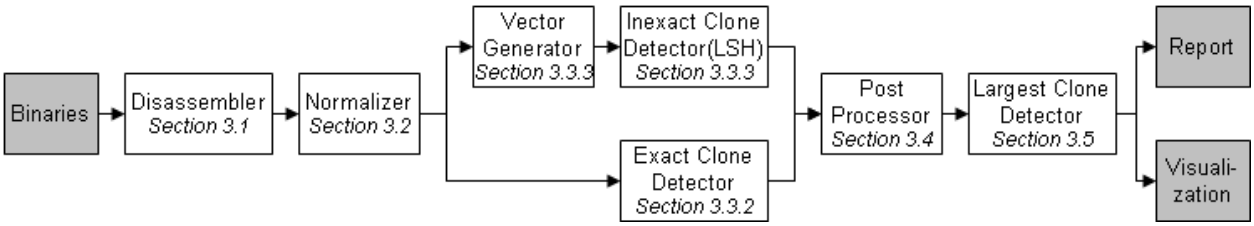

FIGURE 2.1. Disassembly and clone detection process.

Low-level binaries offer additional interesting challenges for clone detection. First, the problem demands better scalability because a single source statement is normally compiled down to many assembly instructions. Second, various choices made by a compiler, such as register and storage allocation, complicate detection. To see this, consider the following IA-32 assembly code:

```
mov [0x805b634], 0x0
mov [0x805b63c], eax
add esp, 0x10
mov eax, ebx
```

where `[0x805b634]` dereferences memory location `0x805b634` (similarly for `[0x805b63c]`), and `eax`, `ebx`, and `esp` are registers. If we use the specific memory addresses or register names for clone detection, we will likely to be too specific and miss *true clones*. On the other hand, if we simply use opcodes (*i.e.*, mnemonics) of the instructions, we will likely to be too general and report *false clones*. Third, assembly instructions have a fixed, almost flat structure, while source programs can have arbitrarily deep structures. The rich structural information in source code is a key factor allowing source-level clone detectors, such as Deckard [46], to perform well. All these differences require novel techniques for detecting binary clones.

In this chapter, we present the first practical binary clone detection algorithm. Our algorithm follows a general tree similarity framework [46]: instead of performing a quadratic number of pair-wise comparisons of instruction sequences, it models the essential structural information of the instruction sequences with numerical vectors and groups *similar* vectors to identify clones. We present novel techniques to generate precise and robust vectors for binaries and to compactly represent the vectors for improved scalability.

We have implemented our algorithm and evaluated it on Windows XP system binaries with a total of 50 million instructions. The results indicate that our technique is both scalable and precise. All the Windows XP system binaries can be processed routinely in under a few hours, and the detected clones are accurate with few false positives. Roughly 20% of the code appears in at least one clone cluster, which is consistent with results for source code [46, 48, 56]. We also evaluate the correspondence of source and binary clones on the

Linux kernel, demonstrating that the binary clones our tool detects typically are caused by real clones in the source code. To better understand the potential of our technique, we also consider the impact of compiler optimizations on our results (see Section 2.5).

The rest of the chapter is structured as follows. We first provide a high-level overview of our algorithm (Section 2.2). The detailed algorithm is presented in Section 2.3. Next, we discuss the implementation (Section 2.4) and evaluation (Section 2.5) of our algorithm. Finally, we survey related work (Section 3.6) and conclude (Section 2.7).

2.2. Overview

Figure 2.1 shows the flowchart for our clone detection algorithm. This section explains the process with the help of the simple example from Section 4.1. Detailed technical descriptions of the steps are be given in the corresponding sections shown in the figure.

First, we use a disassembler to process all input binaries and create their intermediate representations. For example, Figure 2.2 shows how we represent the sample instruction sequence from Section 4.1. Notice that each assembly instruction consists of a *mnemonic* (*e.g.*, "`mov`") and an *operand list* (*e.g.*, "`esp, 0x10`"). Our intermediate representation preserves all binary file information, including instructions, functions, header information, segments, etc. Section 2.3.1 describes the representation in more detail.

Second, the normalization step (cf. Section 2.3.2) creates a *normalized* instruction sequence, abstracting away memory- and register-specific information. The following shows the normalized instruction sequence for the example:

```
mov MEM1, VAL1
mov MEM2, REG1
add REG2, VAL2
mov REG1, REG3
```

Third, we perform clone detection on the normalized instruction sequences. We separate the problem into two cases, mostly for efficiency reasons. One case is *exact clone detection*, where only identical normalized instruction sequences are returned. The other case is *inexact clone detection*, where certain differences are tolerated. This is a computationally challenging problem. We use *feature vectors* to approximate structural characteristics of the given assembly instruction sequences and group similar vectors to find clones. See Sections 2.3.3.2 and 2.3.3.3 for more details.

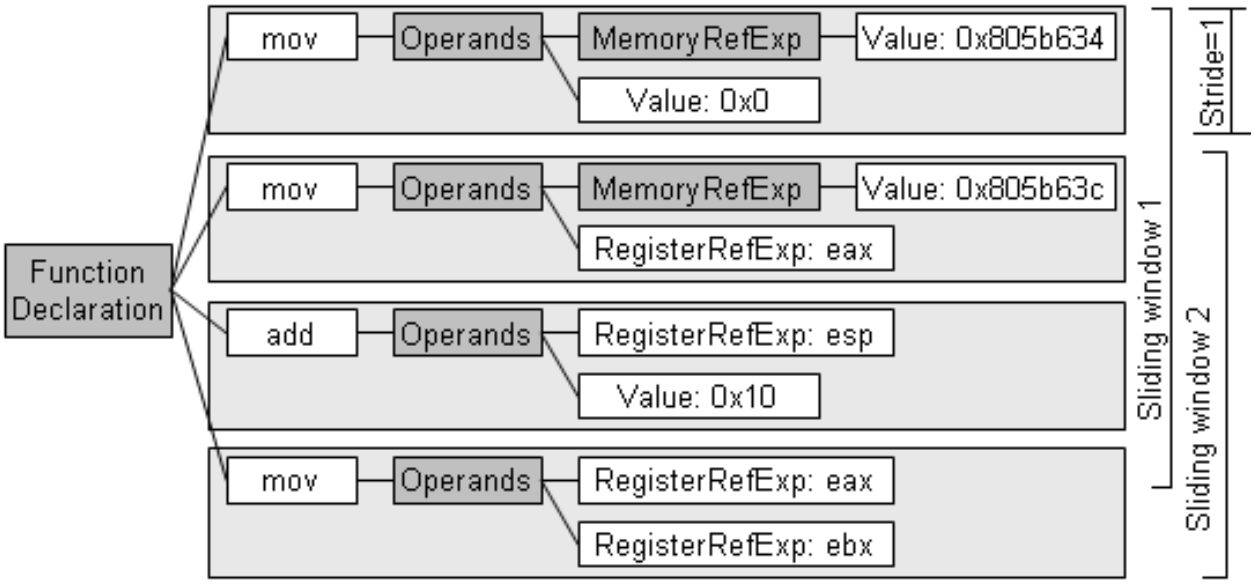

FIGURE 2.2. Example intermediate representation of disassembled binary for stride 1 and window size 3.

We now have a set of *clone clusters*, *i.e.*, instruction sequences of a certain size[1] that are considered similar. For inexact clone detection, the similarity threshold is user-defined while that between instruction sequences for exact clone detection is one. For instance, the clone cluster $C = \{seq_1, seq_2, seq_3\}$ contains three similar instruction sequences, seq_1, seq_2, and seq_3. Two sequences within a clone cluster, say seq_1 and seq_2, may overlap substantially and should not be considered clones. Removing such spurious clones is conducted by a postprocessing step (Section 2.3.4).

So far in the detection process, we consider only instruction sequences of a certain predefined length, but reporting many small clones is not as useful as reporting a few large ones. So, in the final step, we combine smaller contiguous clones into larger ones. The algorithm for doing this is presented in Section 2.3.5.

2.3. Algorithm Description

This section gives a detailed description of our algorithm, structured according to the flowchart in Figure 2.1.

2.3.1. Binary Disassembly. An assembly instruction is a pair of a *mnemonic m* and a list of *operands o*. The mnemonic m represents the particular operation that the instruction performs, and is from a finite set M of possible mnemonics. The list of operands is a variable-length, but typically short, sequence of elements from the set O of possible operands. We partition the set O of operands into three categories: memory references

[1]This is referred to as the *window* as shown in Figure 2.2.

(*e.g.*, "`[0x805b634]`"), register references (*e.g.*, "`eax`"), and constant values (*e.g.*, "`0x10`"). We do not use any of the structure of the individual operands other than this categorization, but we do assume the ability to compare two operands for syntactic equality. In the following algorithm descriptions, an instruction is defined as an element of the set $M \times O^*$, with O partitioned into O_{mem}, O_{reg}, and O_{val}. We use the functions *mnemonic* and *operands* to access the two parts of an instruction, and zero-based subscripts (of either single elements or intervals) to indicate accesses to elements or contiguous subsequences of a sequence; the $+\!\!+$ operator is used to indicate sequence concatenation, and the $+$ operator is used to add a single element to a set or bag. The function *type* maps from O to the set $OPTYPE = \{MEM, REG, VAL\}$ based on the particular category of an operand.

The full disassembly of a particular executable or library is defined as a sequence of functions, with each function containing a sequence of instructions. We define clones in terms of *code regions*, which are simply contiguous subsequences of the instructions of a single function, along with information on the starting address, function, and file of that list of instructions. We ignore that extra information for clone detection, but it is preserved by our algorithms and used by the postprocessing and visualization stages. We assume that algorithms that create and/or transform code regions implicitly process the extra information appropriately. When the distinction is important, the two functions *instructions* and *extraInfo* access the two parts of a code region. The actual process of creating the disassembled instructions for a program and grouping them into functions is implementation-specific; our particular implementation is explained in Section 2.4.

We split each function of a binary into code regions using two parameters *window* and *stride*. The window is the length of the code region to generate. The starting points of the code regions within a function are separated by the stride; note that code regions can, and almost always will, overlap. For example, with window size 50 and stride 10, the first three code regions contain instructions 0–49, 10–59, and 20–69 respectively. Algorithm 1 is used to compute the code regions within a particular function. As the window size and stride are constant for a particular run, this algorithm takes linear time in the number of vectors produced. For a single function F of length l, the number of vectors generated is $\lfloor (l-w+1)/s \rfloor$.

2.3.2. Code Region Normalization. As explained in Section 2.2, the particular operands used in a code region may be specific to that code region, and so some "fuzziness" should be allowed in clones. For example, two regions may be identical except for certain constant values, offsets in memory locations, or particular addresses used as branch targets. In order to account for these differences, we normalize the instructions in a code region using Algorithm 2. This function takes a list of instructions in the $\langle mnemonic, operands \rangle$ format and converts them to *abstract instructions* of the form $\langle mnemonic, abstract\ operands \rangle$. An *abstract*

Algorithm 1 Generate code regions.

Require: f: Disassembled instructions for a single function
w: Window size
s: Stride
Ensure: R: The set of code regions
1: $R \leftarrow \varnothing$
2: **for** $i = 0$ **to** $length(f) - w$ **step** s **do**
3: $thisRegion \leftarrow f_{[i,i+w)}$
4: $R \leftarrow R + thisRegion$
5: **end for**

Algorithm 2 Normalize a code region.

Require: r: Input code region
Ensure: r': Output abstract code region
/* N is a mapping from *OPTYPE* to the sequence of operands of that type seen so far */
1: $N \leftarrow \varnothing$
2: $r' \leftarrow \langle\rangle$ **with extra info** $extraInfo(r)$
3: **for all instructions** i **in** r **do**
4: $ops' \leftarrow \langle\rangle$
5: **for all operands** o **in** $operands(i)$ **do**
6: $t \leftarrow type(o)$
7: **if** o **is an element of** $N[t]$ **then**
8: $idx \leftarrow$ **zero-based index of** o **in** $N[t]$
9: **else**
10: $idx \leftarrow length(N[t])$
11: $N[t] \leftarrow N[t] +\!\!+ \langle o \rangle$
12: **end if**
13: $o' \leftarrow \langle t, idx \rangle$
14: $ops' \leftarrow ops' +\!\!+ \langle o' \rangle$
15: **end for**
16: $i' \leftarrow \langle mnemonic(i), ops' \rangle$
17: $r' \leftarrow r' +\!\!+ \langle i' \rangle$
18: **end for**

operand is a pair of an operand type (either *MEM*, *REG*, or *VAL* from the set *OPTYPE*) and a natural number indicating the index of the first occurrence of that particular operand expression within the code region. The operands are numbered separately for each operand type. The normalization produces an *abstract code region*, which is just a list of abstract instructions. The normalization algorithm takes linear time in the number of instructions in the code region, and is run once on each code region in the program.

2.3.3. Clone Detection. We define two ways to find clones among binaries: exact matching of normalized code regions, and inexact matching of feature vectors representing important aspects of the code regions. Both of these algorithms use linear time and space to find the initial set of clone clusters.

Algorithm 3 Find exact clone clusters.

Require: R: Set of abstract code regions
Ensure: C: Set of clone clusters, each of which is a set of code regions

1: $H \leftarrow$ **empty hash table mapping from sequences of abstract instructions to sets of code regions**
2: **for all code regions** r **in** R **do**
3: **add** r **to** $H[instructions(r)]$
4: **end for**
5: $C \leftarrow \{c \in values(H) : |c| \geq 2\}$

2.3.3.1. *Definitions of Clone Pairs and Clusters.* Code regions can appear in clone pairs and in clone clusters. A *clone pair* is an unordered pair of code regions that are "close enough" (by a metric defined later) to be considered to match. We form clone pairs into *clone clusters* by finding groups of clone pairs that all contain the same normalized instructions. For exact matching, and in practice for inexact matching, the clone pair relation is transitive, and so choosing the neighbors of an arbitrary code region is appropriate.

For measuring the accuracy of our clone detection algorithm, we define false positives and false negatives. A clone pair is considered to be a *false positive* when it is found by the clone detection algorithm and yet the normalized instruction sequences of the two code regions in the pair are not identical. A clone pair is a *false negative* if it satisfies the definition of a clone pair given above and yet is not found by our algorithm. False positives and false negatives can never appear when using exact matching of normalized instruction sequences (by definition), but our inexact matching algorithm has both types of error. In order to test the accuracy of our algorithm, we test the inexact matching algorithm with a distance of less than one to simulate exact matching on feature vectors, and determine how well those results match the actual exact matching algorithm. Note that two distinct normalized instruction sequences may have exactly the same feature vector, so there can be false positives in a vector-based matching algorithm even when exact matches of vectors are found.

2.3.3.2. *Exact Clone Detection.* Exact matching uses a traditional hash table on the normalized instruction sequences, as shown in Algorithm 3. Although this algorithm produces a set of clone clusters, the corresponding set of clone pairs can be found by converting the partition C into an equivalence relation. Algorithm 3 requires linear time, and produces exactly the correct set of clone clusters (with neither false positives nor false negatives).

2.3.3.3. *Inexact Clone Detection.* To find inexact clones, we adapt the basic approach developed by Jiang *et al.* [46] for locating source code clones. We characterize each code region using a set F of *features*, each of which identifies one property we consider important. For example, each possible instruction mnemonic is a

feature, and each combination of the instruction's mnemonic and the type of the instruction's first operand is a feature. The features we use are local to each abstract instruction, and can thus be evaluated independently on the instructions in a code region. We count the number of occurrences of each feature within a code region, producing a *feature vector* for the region. Formally, a feature vector is a vector of natural numbers of length $|F|$, based on a fixed but arbitrary order of the features in F. We allow feature vectors to be indexed directly by features rather than requiring an explicit mapping from features to vector indices.

We count the following features of an abstract instruction or code region; F is the disjoint union of the sets given. These five categories of features were necessary to include in the feature vectors to accurately characterize the code:

- M, representing the mnemonic of the instruction;
- $OPTYPE$, representing the type of each operand in an instruction;
- $M \times OPTYPE$, representing the combination of the mnemonic and the type of the first operand when one is present;
- $OPTYPE \times OPTYPE$, representing the types of the first and second operands, in that order, of an instruction with at least two operands; and
- $OPTYPE \times \mathbb{N}_k$, representing each normalized operand with an index under a chosen limit k.

As we treat the window size as a constant, and the number of operands in an instruction is at most a small number, we can treat vector generation for a single code region as a constant-time operation. The overall vector generation for a large set of code regions is then a linear-time operation (each region can be processed independently). Algorithm 4 produces the feature vector for a code region. The *bagToVector* function creates a vector from a bag by counting the number of occurrences of each element of F in the bag.

Once code regions have been mapped to feature vectors, we then define the distance between two code regions as the ℓ_1 distance between their corresponding feature vectors. The distance between the vectors is intended to approximate the dissimilarity between the code regions. We then define an inexact clone pair for a distance δ as an unordered pair of code regions $\{r_1, r_2\}$, with feature vectors v_1 and v_2 respectively, where $||v_1 - v_2||_1 \leq \delta$. The parameter δ affects the similarity required between the feature vectors: having $\delta < 1$ requires the vectors to be identical and thus the code regions to be almost the same, while larger distances allow more dissimilar code regions to appear in clone pairs.

Given the space of vectors and a distance metric, we would like an efficient way to find all vectors within a given distance from a query vector; *i.e.*, we would like a *near neighbor* data structure and algorithm. We follow the approach in Deckard [46] and use *locality-sensitive hashing (LSH)* for this purpose [43]. In

Algorithm 4 *regionToVector*: Generate feature vector.

Require: r: Abstract code region
Ensure: v: Feature vector

$b \leftarrow$ **an empty bag (multiset) of features from** F
for all instructions i **in** $instructions(r)$ **do**
  $b \leftarrow b + mnemonic(i)$
  for all $\langle t, idx \rangle \in operands(i)$ **do**
    if $idx < k$ **then**
      $b \leftarrow b + \langle t, idx \rangle$
    end if
    $b \leftarrow b + t$
  end for
  $ops \leftarrow operands(i)$
  if $length(ops) \geq 1$ **then**
    $b \leftarrow b + \langle mnemonic(i), type(ops_0) \rangle$
  end if
  if $length(ops) \geq 2$ **then**
    $b \leftarrow b + \langle type(ops_0), type(ops_1) \rangle$
  end if
end for
$v \leftarrow bagToVector(b)$

particular, we use the set of hash functions from [35] for the ℓ_1 distance on vectors of natural numbers. LSH is an approximate algorithm, allowing false negatives in order to achieve constant time and space insertion and queries for distance-based matching when given appropriate parameter choices. Our inexact clone detection approach is shown in Algorithm 5. This algorithm requires an empty hash table set to be passed as its input. LSH uses two parameters to create the hash function, and they must be chosen carefully for good performance and accuracy; see Section 2.4.2 for more information. The function $vectorsInBucket(v_1)$ used in the code finds all vectors that hash into the same bucket as v_1 in any of the hash tables in the LSH hash table structure; this set is the same as the set of elements that would be searched in a near neighbor query for the element v_1 in H. Note that this algorithm produces clusters greedily in such a way that each code region is in at most one cluster. Although the order of iteration through the regions can change the set of clusters produced for non-transitive neighbor relations, we assume, supported by past literature, that the relation is transitive or close to it. Allowing overlapping clone clusters can lead to an algorithm requiring quadratic time, while the limitation to non-overlapping clusters uses only linear time.

Although Deckard [46] uses Euclidean (ℓ_2) distances to detect inexact clones in source code, we chose to use ℓ_1 distances instead: our data sets are very different from those used by Deckard and required us to recompute parameters for every group, which could be done analytically for the ℓ_1 norm more easily than for ℓ_2. Every inexact clone detection phase is optimized by first performing exact clone detection and thereby

Algorithm 5 Find inexact clones.

Require: R: Set of abstract code regions
δ: Distance to use for queries
H: Empty LSH hash table set
Ensure: C: Set of clone clusters
1: $V \leftarrow \{regionToVector(r) : r \in R\}$
2: **for all vectors** v **in** V **do**
3: **insert** v **into** H
4: **end for**
5: $C \leftarrow \varnothing$
6: **for all vectors** v_1 **in** V **do**
7: **if** $v_1 \notin \bigcup C$ **then**
8: $M \leftarrow vectorsInBucket(v_1)$
9: $c \leftarrow \{v_2 \in M - \bigcup C : ||v_2 - v_1||_1 \leq \delta\}$
10: $C \leftarrow C + c$
11: **end if**
12: **end for**
13: $C \leftarrow \{c \in C : |c| \geq 2\}$

making sure that every distinct vector is only represented once when doing inexact clone detection, even if that same vector is used for several code regions. The results of the exact and inexact clone detection are merged when both runs are finished.

2.3.4. Removing Trivial Clones. When the stride s used to generate code regions is smaller than the window size w (*i.e.*, the length of each code region), it is possible that two code regions in the same function almost completely overlap with each other. As the feature vector generation is almost independent of the order of the instructions, it is likely that these two regions would be detected as a clone pair. However, such a pair is not interesting as it is effectively stating that a region of code is a clone of itself. We define a parameter o that indicates the fraction of instructions that two regions can have in common and still be considered a legitimate clone pair, and reduce the set of clone clusters using Algorithm 6.

It is unclear what it means when two overlapping code regions r and r' are considered clones. In our experiments we have decided on an overlap of 50 percent or less. Although a few of the clones that are not filtered out can be considered false clones, we are more concerned with completeness of our clone-set than this problem. If it is desirable to do so, filtering out all overlapping regions can be done just by changing o.

This algorithm is finding the maximum independent set of an interval graph (the graph of overlapping vectors within a single clone cluster and function), a problem that can be solved using sorting in $O(n \log n)$ time and $O(n)$ space. Here, the nonlinear term is only applied to those code regions that are both within the same clone cluster and the same function, making the algorithm effectively linear in practice.

Algorithm 6 Remove trivial clones.

Require: C: Set of clone clusters
o: Allowed fraction of overlap
w: Window size
Ensure: C': Post-processed set of clone clusters

$C' \leftarrow \varnothing$
for all clusters c **in** C **do**
  $c' \leftarrow \varnothing$
  for all functions f **containing regions in** c **do**
    $R \leftarrow$ **code regions from** f **in** c
    sort R **by instruction offset within** f
    $first \leftarrow \top$
    $lastOffset \leftarrow 0$
    for all code regions r **in** R **do**
      $offset \leftarrow$ **offset of** r **within** f
      if $first$ **or** $offset \geq lastOffset + o \cdot w$ **then**
        $c' \leftarrow c' \cup \{r\}$
        $lastOffset \leftarrow offset$
      end if
      $first \leftarrow \bot$
    end for
  end for
  if $|c'| \geq 2$ **then**
    $C' \leftarrow C' + c'$
  end if
end for

2.3.5. Finding Largest Clones. Finding the largest sequences of instructions A and B that are part of a clone pair $\{A,B\}$ is important to avoid an overestimate in the reported number of clones. Since there are overlapping vectors A and A' in the set of vectors in C, we must expect that the number of clones and the total number of vectors in the clones are overestimates. For instance, if there is a sequence of length n ($n \geq w$) that matches between functions f_1 and f_2 then it will be represented by up to $\lfloor (n-w)/s \rfloor + 1$ clusters in C. If n is 500, the window size w is 40, and the stride s is 1, that single logical clone pair becomes 461 pairs in the output.

Algorithm 7 finds the largest clone pairs $\{a,b\}$, but does not find the largest clone clusters. Clone pairs can be generated from a clone cluster by taking all unordered pairs of distinct code regions from the cluster. There may be a quadratic number of clone pairs from a single clone cluster, although clusters are typically not large. The algorithm relies on a total order among code regions; the code regions are first ordered by the functions containing them, and then by the instruction offset within each function. The algorithm assumes that a is less than b in each clone pair. We sort the clone pairs according to the two elements of the pair in

Algorithm 7 Estimate the largest clone pairs.

Require: L: Sequence of clone pairs
Ensure: L: Set of largest clone pairs
1: **sort** L **using the order in the text**
2: $n \leftarrow \varnothing$
3: $L' \leftarrow \{\varnothing\}$
4: **for all** n' **in** L **do**
5: **if** $overlap(n,n')$ **then**
6: $n \leftarrow join(n,n')$
7: **else**
8: $L' \leftarrow L' + n$
9: $n \leftarrow n'$
10: **end if**
11: **end for**
12: $L \leftarrow (L' + n) - \{\varnothing\}$

that order. Sorting ensures that if there are two sequences in A and B that overlap then all clone pairs for those sequences will be adjacent in the list.

Given a sorted list we do a linear search over the list of clone pairs where clones in sequence that overlap are joined into a larger clone pair. We define two clone pairs $\{a,b\}$ and $\{a',b'\}$ to be overlapping if the code sequence corresponding to a overlaps with a' and b overlaps with b'.

When joined, the two clone pairs are then replaced by a larger (covering more code) clone pair. The joining is inherently optimistic and assumes that if two overlapping clone pairs are joined then the joined pair is also a clone. We expect this to create false positives, but since the number of those larger clones is small, it is cheap to run a more expensive algorithm on the joined pairs in order to remove false positives.

The computational complexity of Algorithm 7 is $O(n \log n)$ where n is the number of clone pairs, as sorting is the most computationally complex task. The number of clone pairs is itself $O(m^2 N)$ in the worst case, where N is the number of clone clusters, and m is the size of the largest clone cluster. The actual number, however, is proportional to the sum of the squares of the clone cluster sizes, and the number of large clone clusters is expected to be small.

2.4. Implementation

We implemented our algorithms in a clone detection tool that uses IDA Pro [1] for disassembly and is therefore capable of interpreting both ELF (Linux) and PE (Windows) executable formats. Although in this study we selected IDA Pro as a front-end for disassembling the binaries and locating functions within them, our implementation does not rely on it.

Our clone detection and analysis system is implemented in C++ and uses a SQLite (version 3) database for communication. Each phase of the analysis is implemented as a separate program, allowing new analyses to be added modularly. The first pass converts a set of disassembled functions and instructions from IDA Pro into the ROSE intermediate representation [72], and from there to both feature vectors and abstract assembly instructions, which are inserted into the database. Based on the database, either exact or inexact clone detection may be run to produce a new table of clone clusters, which may be operated on by post-processing, largest clone detection, or clone visualization.

2.4.1. Memory and Computational Efficiency. The dimensionality of our feature vectors is 26 times larger for binaries than for the vectors used for source code in Deckard [46]. The memory usage and computational complexity of LSH thus increase by at least 26 times when using LSH on object code as compared to source code. Since each dimension takes at least one byte, and often more, it is necessary to create a compressed representation for large data sets.

We made the observation that our feature vectors are sparse and largely consist of small numbers. For example, we define a large set of features F that includes features such as the number of references to the 80th memory reference in a code region; it is unlikely that there are 80 memory references in a region, and so this element of the feature vector is almost always zero. Since we only generate the data set once and use it many times, it is beneficial to construct a compressed representation once and reuse it, saving disk and memory space. Because zero elements in the vector are handled specially, they can also be skipped in some computations to save CPU time.

We use a compressed representation that run-length encodes contiguous sequences of zero elements in the vector, plus encodes numbers using variable numbers of bytes based on the values of the particular entries. Several contiguous vector elements that are each near zero can also be packed into a single byte. Experimentally, we have shown that this technique can use 17 to 36 times less space to store the same set of vectors. Generally, computation time is traded for memory when using compressed representations, but we are able to operate on the compressed vectors directly and thus take advantage of the fact that many vector elements are always zero to save computation in the vector kernels used by LSH (dot products, element extraction, norm computation, etc.), as well as not requiring time or storage for decompression. We store vectors in compressed form both on disk and in memory. Without compressing the vectors, our data sets would require hundreds of gigabytes of disk space; compression allows the same data sets to be processed entirely in memory if desired.

2.4.2. LSH Parameter Tuning. If a family of LSH hash functions H is used to find clones in $O(n \log n)$ time then the parameters controlling the probabilities of two similar elements colliding must be carefully chosen [5]. An LSH data structure consists of l independent hash tables, each containing the same data but a different hash function; each hash function is built from k components. These two parameters determine the runtime and accuracy of the algorithm. The general rule is that a larger k increases the false negative rate while a larger l increases the collision rate (*i.e.*, the percentage of elements scanned in the query but that are not within the desired distance). LSH's memory usage is thus proportional to l, even with the number of buckets and bucket size fixed. The k parameter does not affect memory usage substantially, and has only a minor impact on the time used for hash table operations; however, the value of k determines how many results are returned for a given query, leading either to unnecessary, failing distance tests or false negatives.

Parameter selection was challenging for our dataset since the dimensionality of our dataset is 26 times larger than in Deckard [46]. Our dataset has large distances between different vectors, and in particular the ℓ_1 norms of the vectors in our data set vary. We observed through experiments that LSH does not handle such data sets well, and thus we apply LSH separately to sets of vectors grouped using their ℓ_1 norms, as is done in Deckard [46]. Groups are chosen to overlap such that any two vectors that are within the distance bound δ are in at least one group together. We also, based on Deckard [46], use similarity values as a measure of the level of matching between clones; it is converted to a distance bound for each group based on that group's smallest ℓ_1 norm. We then choose each group's LSH parameters individually.

The experimental approach for selecting parameters as done in [4] is not viable for our application, and so we choose optimal parameters for LSH analytically using the approach from [77]. We use their model of LSH behavior to define a function from k, l, and the distance d between two vectors to find the probability of one being found in a query for the other. Assuming a uniform distribution of distances between vectors, we add the probability that vectors within the distance bound will not be found (false negatives) and the probability that vectors outside it will be found (collision rate). This sum provides a score for that particular set of parameters. We then use the cost model from [77] to estimate the time used for the given set of parameters, and vary k to optimize the cost for a given level of accuracy. We find l using a formula given in [77]; we can choose l to achieve an arbitrarily low false negative rate.

2.4.3. Experimental Setup. We performed a large scale study on our clone detection tool using the disassembled representations of the Windows XP system executables and libraries. All runs were done on a workstation with two Xeon X5355 2.66 GHz quad-core processors and 16 GiB of RAM, of which we use one core for our experiments. We have a 4-disk RAID with 15,000 RPM 300 GB disks. The machine runs

Window Size	# of files	# of functions
500	863	7,072
200	1,486	42,819
120	1,633	97,588
80	1,681	168,224
40	1,722	342,874

TABLE 2.1. Functions and files with ≥ 1 code region.

Window Size	Vectors	Clusters	Clones
500	2,588,507	206,785	704,263
200	7,963,384	587,582	2,039,093
120	13,130,524	966,604	3,419,038
80	18,304,493	382,023	1,227,669
40	27,946,044	2,368,355	8,636,593

TABLE 2.2. Clone statistics.

Red Hat Enterprise Linux 4 with kernel version 2.6.9-78. None of our runs used more than 4 GiB of memory for inexact clone detection; we do not apply grouping for exact clone detection, and keep all of the data in memory simultaneously, but applying grouping would be trivial.

2.5. Experimental Results

In this section we evaluate our tool using a large-scale study on the Windows XP system libraries for various window sizes. According to Table 2.1, there are many small functions that do not contain any code regions larger than length 40. Approximately 2/3 of the 1,108,535 functions in Windows XP system files have fewer than 40 instructions, and thus cannot be tested for clones using that window size. Reducing the window size would allow these functions to be covered by the algorithm. There are relatively few files, however, that have functions with 40 instructions but do not have any functions with 200 instructions.

2.5.1. Clone Quantity. For a range of different window sizes, we evaluated how many code regions, clone clusters, and clones (code regions in a clone cluster) are produced by our algorithm; this data is shown in Table 2.2. We also used code coverage by clones (the percentage of the original instructions that are in at least one clone) as a measure of clone quantity.

Figure 2.3 shows that the code covered increases with decreasing similarity for all window sizes, but the total coverage is much larger for smaller window sizes. The total coverage decreases with increasing window size because many smaller functions do not contain enough instructions to fill one code region,

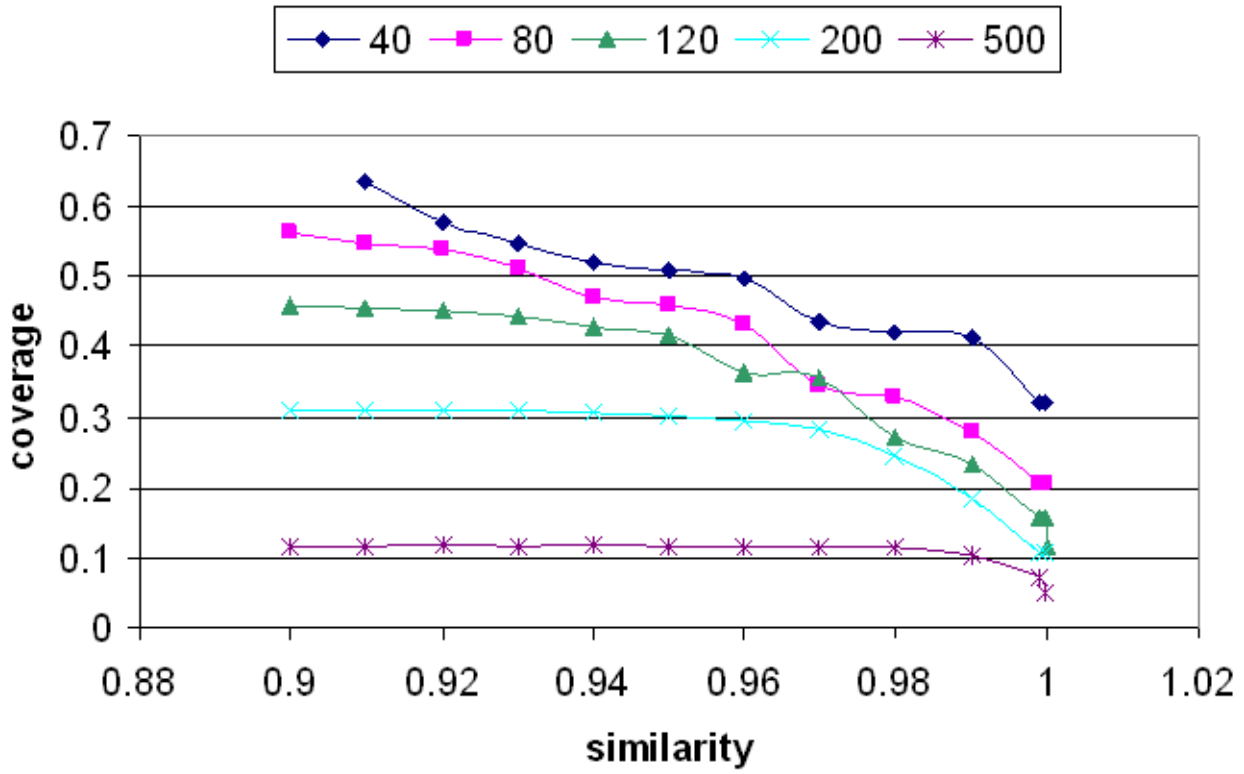

FIGURE 2.3. Total clone coverage.

as each must be within a single function; also, smaller regions of code may be clones even when they are contained in larger regions that are not. When the similarity is decreased, the amount of code covered by clones tends toward the amount covered by code regions. For instance, the maximum possible coverage is 12% for window size 500 and 32% for window size 200 for any similarity.

Window Size	> 2	> 4	> 16	> 64	> 128
500	69,775	16,705	2,420	531	0
200	196,540	59,336	5,694	905	11
120	325,010	105,773	10,007	1,404	125
80	467,737	157,983	15,442	2,117	337
40	798,272	286,066	32,585	3,919	890

TABLE 2.3. Sizes of clone clusters.

Table 2.3 shows that when the window sizes increase the average size of the clone clusters decreases. This validates the intuition that larger instruction sequences are more unique than shorter instruction sequences.

2.5.2. Clone Quality. We evaluate two different aspects of clone quality. First, we evaluate the quality of the binary clones relative to source code clones on Linux kernel 2.6.27-9. Second, we evaluate the accuracy

of the clones found using the ℓ_1 norm compared with those found by plain hashing of the normalized assembly instructions.

During the steps from parsing the source code to assembly the compiler performs several steps that transform the code using optimization heuristics, and some compilers normalize the parsed representation of multiple languages into a common intermediate representation (*e.g.*, GCC). For many applications it is important that there is a mapping from the binary clones to the source code. Clones caused by these compiler artifacts cause two functions to be related by clones in the binary when they are not considered clones in the source code. It is also possible, but less likely, that clones in the source code do not appear as clones in the binary; this case happens when the contexts of the two code sections are different enough that they are optimized differently.

To evaluate the quality of the binary clone pairs, we inspect the source code of their containing functions. If there is a clone relationship between the functions in the source code, we decide that it is a true clone. For this particular study, we chose to evaluate the binary clones found in the Linux kernel compiled with default compiler flags (`-O2` plus extra optimizations).

Table 2.4 shows the result of a human oracle inspecting the binary clones from exact clone detection runs using various window sizes. No matter how many clones there are that connect two functions, they are still represented by one number in the table. The function clones are categorized by the cause of the binary clone, where "trivial" means that the binary clone corresponds to a source code clone.

For complicated functions it is hard to determine if a clone is caused by inlining, a macro call, or compiler artifacts so we classify the cause as unknown. All the unknown clones will in the worst case be caused by compiler artifacts, but this is probably not true in general.

Window Size	80	200	500
Trivial	166	20	2
Inlining	15	1	0
Macro	29	0	0
Same source	93	30	9
Unknown	29	3	0
Total	332	54	11

TABLE 2.4. Manual analysis of the quality of exact code clones for Linux kernel 2.6.27-9 optimized with default flags (`-O2`).

The Linux kernel uses kernel modules for drivers. Many drivers are created using code copying, and in some cases where two kernel modules are not meant to be loaded at the same time they partly share source code. These kernel modules will have functions that correspond to the same function in the source code.

Window Size	Mismatch Rate (%)	Coverage (%)
500	3.3	3.0
200	2.9	7.9
120	2.9	12.5
80	2.9	17.1
40	3.2	27.8

TABLE 2.5. Clones not matching between the ℓ_1 norm and exact hashing, and clone coverage.

Second, we evaluate how well binary clones found using exact matching on feature vectors match clones found using exact hashing on the normalized instruction sequences. We define a mismatch as any code region that is in a clone cluster (as defined by exact matching of feature vectors) but has a different normalized instruction sequence from the plurality of code regions in that cluster; this metric is simply counting all mismatched clone pairs (as defined in Section 2.3.3.1) between each clone within the cluster and a single element having the plurality instruction sequence. This metric shows how well feature vectors characterize normalized instruction sequences. Manual inspection of the clones found by our inexact clone detection found similar sequences of instructions for large window sizes, with less accurate results for smaller window sizes as one expects. We found our mismatch rate to be low as shown in Table 2.5. When only using mnemonics for detecting code clones, rather than the other features we consider, the mismatch rate is high as shown in Table 2.6.

2.5.3. Impact of Compiler Optimization. There is a large body of code available for reuse under open source licenses. These licenses put certain restrictions on how it can be reused. It is for instance not permitted to reuse GPL-licensed code in closed-source projects. Because the source code is typically not available for the violating projects, such license violations can be difficult to detect, and infringing authors feel safe by only making the binaries available.

In general these authors can apply arbitrary obfuscation techniques, but according to `gpl-violations.org` [38], the common practice is that the source code is not changed. Compilation using different compilers and compiler options will usually generate different assembly code from the same source code. We empirically validated this in the following experiment.

Window Size	Mismatch Rate
500	15.54
200	16.92
120	18.31
80	19.19
40	26.08

TABLE 2.6. Mismatch rates using only mnemonics.

The impact of compiler optimizations in terms of binary clones is determined by inspecting the assembly code. For our particular evaluation we chose to compile Linux kernel 2.6.27-9 multiple times using different compiler options. We then look at the same function, compiled with the different sets of options, and test whether that pair share at least one clone. Using specific window sizes, Table 2.7 shows how many functions are considered clones of themselves when compiled with `-O1` and `-O2` using GCC. There are no clones between the code generated with `-Os` and the code generated with `-O1` and `-O2`. This is reasonable since `-Os` optimizes for size while `-O1` and `-O2` optimize for speed. The results confirm that compilers may produce substantially different code with different compiler options.

Similarity	1.0	0.98	# possible functions
80	99	430	13899
200	20	115	4675
500	5	39	1134
1000	1	32	1000

TABLE 2.7. `-O1` vs.`-O2`: Impact of compiler optimization on clone detection for the Linux kernel, window size along y axis.

Given N compilers with M sets of compiler options, there are potentially $N \times M$ different assembly sequences that need to be detected. For projects where the source code is available, this problem can be overcome with a linear increase in clone detection time by compiling the code using available compiler options under different compilers.

2.5.4. Scalability. We show that our tool is scalable by doing a large-scale study on the Windows XP system files for similarities between 0.90 and 1.0, where 1.0 is exact clone detection on normalized instructions and the others are inexact matching on vectors. For all the window sizes used in this study the stride is 1. The vectors are stored in a database for each stride and window size. This database is generated once and reused later. Table 2.8 shows that our vector generation algorithm is scalable.

Window Size	VecGen
500	225
200	247
120	254
80	275
40	277

TABLE 2.8. Vector generation time (minutes).

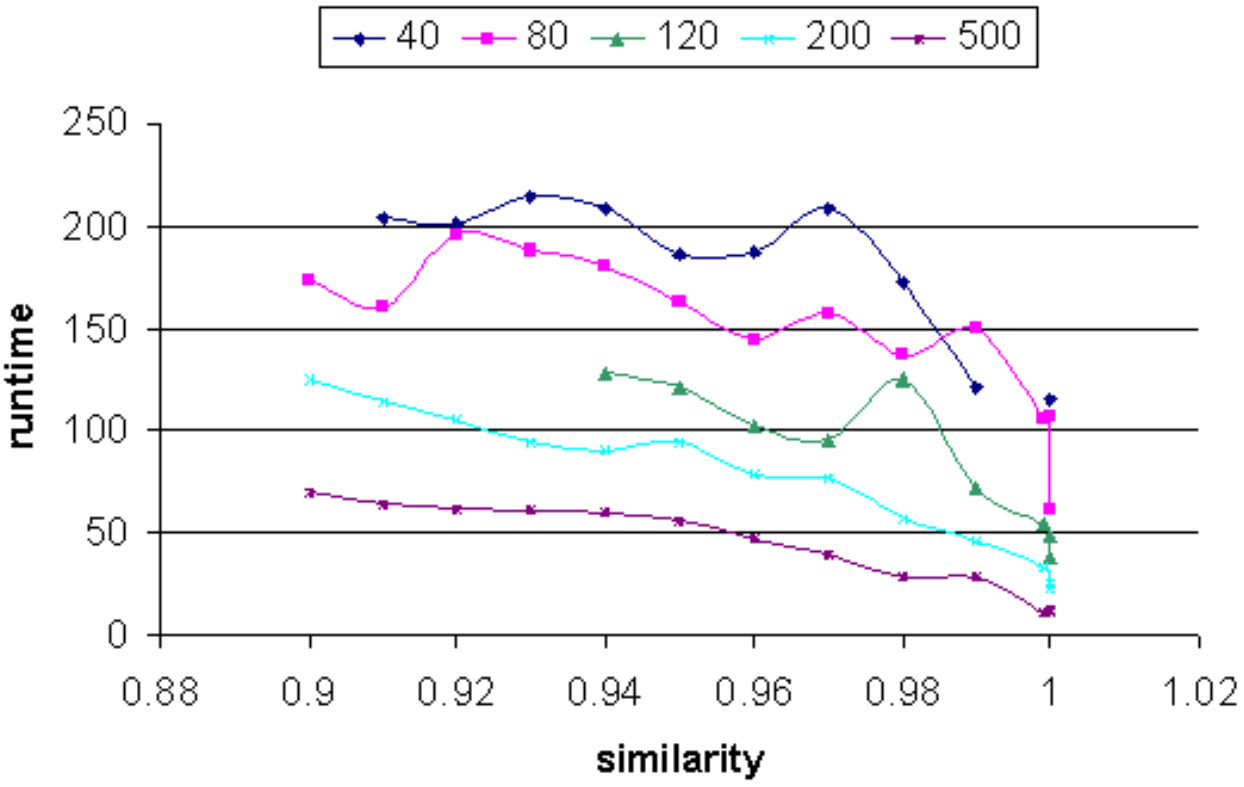

FIGURE 2.4. Clone detection time (in minutes).

Figure 2.4 shows that our tool detects clones scalably on our vector database for all window sizes. The runtime seems to increase as expected for an $O(n \log n)$ algorithm (for n vectors); two factors contribute to an increase in the data set size: decreasing window size leading to more vectors, and decreasing similarity grade leading to a higher likelihood of hashing vectors into the same bucket. The runtime varies for the same window size due to the load from other processes on the machine. Because exact and inexact clone detection use different algorithms, their run times differ; in particular, exact matching uses a much faster algorithm than inexact.

2.5.5. Estimation of Largest Clone Size. Using Algorithm 7, we estimate the largest clones for exact clone detection. Unlike all other algorithms, finding the largest clones produces a set of clone pairs instead of a set of clusters. Table 2.9 shows that smaller window sizes generate clone pairs that combine to form larger clones. Because a data set generated using a small window size covers a smaller part of a larger clone it is expected that the combination of the smaller clone pairs will be an overestimate of the number of larger clones, but the table shows that this is only a slight overestimate for all window sizes except 40.

2.5.6. Visualization of Clone Clusters. Figure 2.5 illustrates our clone detection efforts on Windows XP. Green boxes represent clones and all other colored boxes represent files within the `system32` directory in Windows XP. The height of a clone (green box) represents the number of clones detected between a set of files. The height of a file represents the number of functions in that file that are clones with other functions (contained in other files); *i.e.*, the more clones a file has, the larger the box. The boxes' widths are determined from their labels. Different subdirectories within the `system32` directory are illustrated with different colors. For instance, the `drivers` directory is yellow and the `usmt` directory is orange. The image reveals that much of Windows XP is somewhat related, *i.e.*, many clones exist.

Figure 2.6 shows portions of Figure 2.5 in more detail. For instance, *clone 37* reveals the relationship between Windows VPN components, such as `CVPNDRVA.sys`, `vpnapi.dll`, and `CSGina.dll` (which can use VPN functionality). Similarly, *clone 166* reveals a clone relationship between the different Windows Management Instrumentation components. *Clone 986* is another clone detected by our tool that contains, amongst others, the Windows system information application `systeminfo`. It appears that `systeminfo` shares functionality with the system applications `tasklist`, `taskkill`, and `getmac`. All results illustrated in Figure 2.5 and Figure 2.6 use a window size of 120 for clone detection.

Window Size	> 40	> 80	> 200	> 500	> 1000
500	5,569	5,569	5,569	8,799	5,377
200	89,888	89,888	89,888	8,965	5,406
120	299,933	299,933	83,940	9,113	5,441
80	640,186	640,186	108,766	9,256	5,472
40	2,440,286	931,672	178,871	12,335	5,578

TABLE 2.9. Sizes of largest clone pairs with exact clone detection.

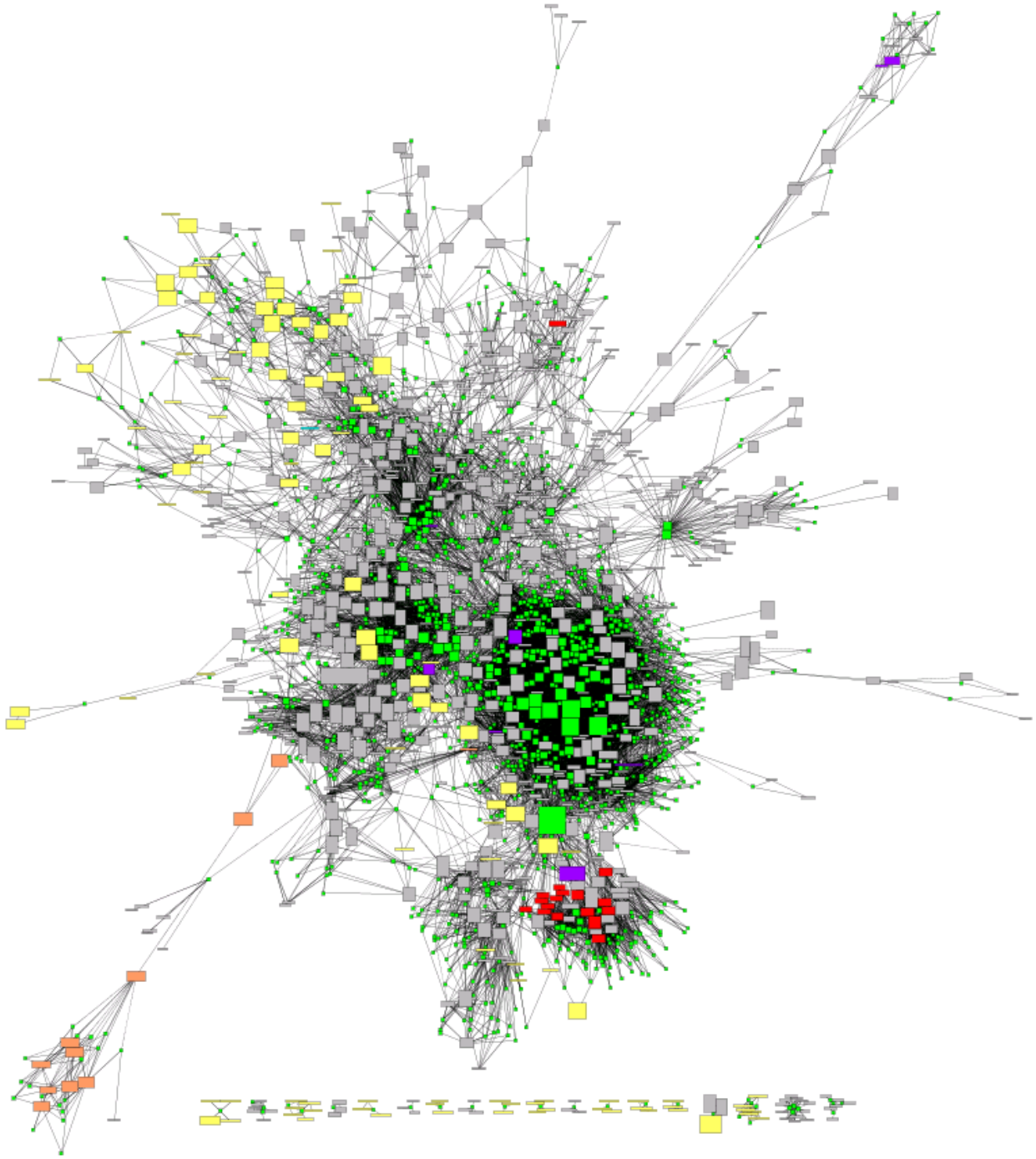

FIGURE 2.5. Clone visualization of Windows XP.

2.6. Related Work

The most closely related work to ours is by Schulman [73] to find duplicated instruction sequences in a large set of binaries. Mnemonics and API calls are used to find binary clones. This approach is inaccurate as

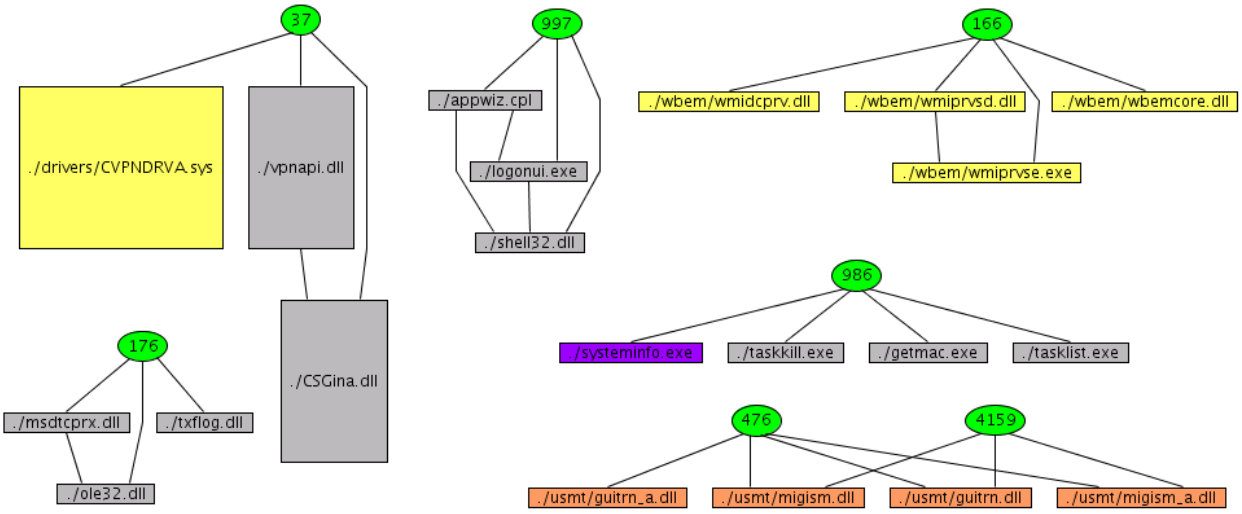

FIGURE 2.6. Visualization of selected clones in Windows XP.

it insufficiently categorizes the instruction sequences in a program. Besides Schulman's work, we are not aware of any previous research on the topic.

Our work is also related to the large body of work on malware detection and analysis [15, 24, 25, 26, 53, 92], where the goal is to decide whether an unknown piece of binary is malicious or not. The common setting is that there are a large number of known malware samples and the problem is to determine whether the unknown malware is a polymorphic or metamorphic variant of any of the known samples. The work can be classified as either *signature-based* (*e.g.*, using regular expression matching of binary code) or *behavior-based* (*e.g.*, using runtime behavior, such as sequences of system calls, for matching). The key difference is that in binary clone analysis, it is not to analyze a single, usually small, binary against a set of other code, but to find all possible matches among a collection of, potentially large, binary code. Thus, the scalability requirement is much greater for binary clone analysis. This chapter provides the first scalable and accurate algorithm for the problem. We do not however consider obfuscations beyond leveraging different compilers or compiler options, and it is an open question whether it is feasible to scale more expensive malware analysis techniques to the setting of binary clone detection.

Our work is also related to source-level clone detection to find duplicated source code. This is a well explored topic, and many scalable and precise tools exist to solve this problem. Some tools are tailored toward finding plagiarism, such as Moss [71] and JPlag [2]. Others are more tailored for software engineering applications such as refactoring and defect detection. Tools such as CP-Miner [56] and CCFinder [48] are token-based and usually more accurate and scalable, but tend to be sensitive to minor code changes. There are also tools based on abstract syntax trees (ASTs) [11, 12, 46, 86]. However, all the above tools are for

source code, while ours is the first practical clone detection algorithm for binaries. In particular, we adapt the framework of Jiang *et al.*'s work on Deckard [46] and introduce precise and compact feature vectors to capture essential structural characteristics of binaries.

2.7. Conclusions

In this chapter, we have presented a novel clone detection algorithm for binaries. We have implemented the algorithm and conducted a large-scale empirical evaluation of it on the system files from Windows XP. Results show that it is scalable and precise, and thus practical to enable many applications on binaries. For future work we plan to conduct further studies with our technique, for example, by analyzing different versions of Windows and other operating systems, and other application components such as Microsoft Office. We also plan to apply our technique to a number of application domains such as detecting latent bugs and a large scale study on protecting intellectual property.

CHAPTER 3

Mixed Binary Analysis for Fine-Grained Software Theft Detection

Software license violations are common. Examples include code a programmer wrote for an ex-employer or open source software licensed under copyleft, such as the Linux kernel reused in a closed source router. Illicit reuse is only cost-effective if undetected. Unfortunately for victims, it is easy to hide stolen source code in binaries, because binary analysis is challenging due to translation, the lack of types, and rewriting, like optimization and obfuscation.

The point of stealing code is to appropriate its behavior, so semantic similarity is unaffected by compiler transformations and obfuscation. Inspired by the Schwartz-Zippel lemma, we compare function output on random input to measure semantic similarity. This measure has surprising specificity, the ability to correctly identify dissimilar functions, given only a few inputs. Despite its effectiveness, it is susceptible to false positives due to cleanroom implementations or functions whose behavior changes at only a few inputs. We show that measuring syntactic, after semantic, similarity increases precision. Because of its reliance on semantic similarity, our approach is robust against any adversary that preserves the semantics of the code it steals.

We present SLEUTH, a hybrid semantic and syntactic detector that advances the state of the art in theft detection in Linux binaries. SLEUTH does not require source, is robust against compiler optimizations and source-level obfuscation and is fine-grained, detecting theft at the granularity of functions. Evaluated on the 10 most popular C programs on GitHub in Fall 2013, SLEUTH achieved a mean precision of 88.12%, even when the compilers and optimization levels that produced a victim and a suspect were unknown.

3.1. Introduction

Developers commonly reuse code to save time, money and effort. Most of the time, this reuse is simply good practice. Code reused in violation of its license, however, is theft. For example, a closed-source program that includes GPL-ed code violates the GPL. Illicit code reuse is easily hidden in binaries, because they are difficult to analyze, and happens all too often [39]. Most proprietary code ships in binary form, which may contain stolen code, hidden behind the binary veil. Although we suspect code theft occurs, theft is seldom

reported. The discovery of illicit code reuse in binaries is the *binary code theft* problem and a main focus of this dissertation.

An unscrupulous developer commits binary code theft when he knowingly copies source code from the victim's code base for use in another code base in violation of the victim's license. Code theft is only cost-effective if it is not detected, but due to the difficulties of binaries it is relatively easy to hide the theft by only distributing the binary form. In general, detecting illegal code replicas is undecidable. Our threat model therefore considers an adversary who preserves the semantics of the code he steals. A particular realization of this adversary steals code to save time and money, so he minimizes the cost of finding and hiding his theft: a thief will not knowingly do more work to steal than to reimplement code. Thus, he steals functions since they are easier to find and transplant, and prefers to hide the theft with tools, notably a compiler, sometimes using non-default optimization.

This cost-benefit analysis restricts the dishonest developer's choice of tools to those that employ conservative, semantics-preserving transformations with limited impact on the stolen code's nonfunctional properties, since it is a waste of time to steal code only to cripple its functionality, such as incurring the overhead of encoding and decoding its output, in order to hide its theft. Thus, the adversary will eschew source code obfuscators that undermine the maintainability of code or binary obfuscators that nontrivially slow down code and will even avoid refactoring tools, since they can create harder to maintain code [85]. Fortunately for the reprobate developer, compilers have been, to date, excellent at hiding code theft in binaries, even without resorting to obfuscation, beyond optimization, because different compilers, including different versions of the same compiler, and different compiler flags generate quite distinct binaries from the same source code.

The same cost benefit reasoning also justifies our focus on function-granular theft. The thief must find code to steal, identify its dependencies, and transplant it into a target. Unlike arbitrary code fragments, functions have signatures that implicitly document some of their behavior. Thus, they are a natural target of a search for code to illicitly reuse. A non-function code snippet, in contrast, lacks a signature and syntactic markers for its start and end. Most of the time, a function's signature captures a large portion of its input dependencies and its effects. Side-effects and global inputs are the exception for functions, but dominate the dependency analysis of non-function fragments, whose signatures must be extracted. Functions define a scope; they can be grafted into their target without worrying about name capture, again with the exception of globals. Non-function fragments, in contrast, often require α-renaming for successful transplant. If a thief does steal non-function code fragments, we will detect those that contain functions. Most previous work on binary theft has been file granular because it eases the search for the code to steal, the identification of

dependencies (since most are contained within the file), and the transplantation of the stolen code, which becomes a file copy.

Two distinct lines of research into software similarity detection exist — syntactic and semantic. Alone, neither solves the binary code theft problem. Semantic software similarity detection characterizes code in terms of how it changes program state. Dynamic software similarity detection observes program execution at various levels of abstraction, such as Wang *et al.* [87] system call sequences or Jiang and Su's input-output values [47]). Wang *et al.*'s work is whole-program, and will be difficult to adapt to function-granular theft, since 60% of functions with at least 100 instructions make no system calls (Section 3.5.4). Jiang and Su's work is more fine-grained than our work, since it cuts up a function into consecutive statements along a path, but operates on source code, not binaries. When used to detect binary code theft, semantic software similarity detection techniques are susceptible to false positives, since different implementations of the same specification are similar by definition.

Syntactic software similarity detection techniques search for repeated sequences and patterns in code and is a well-explored research area: many scalable and precise tools exist to solve this problem [6, 33, 46, 48, 52, 56]. However, the source code is not always available, as with commercial off the shelf (COTS) software. Our previous work introduced the first syntactic binary software similarity detection tool [69]. Myles and Collberg k-grams of instruction sequences [60]. Hemel *et al.* apply normalized compression distance to detect whole program theft with impressive results [39], but do not scale down to function-granular theft in our setting (Section 3.5.4). False negatives bedevil these syntactic approaches when applied to the binary code theft problem (Section 3.5.4). When constructing a binary, compilers make many choices, such as instruction, register and storage allocation; their differing choices undermine accurate syntactic theft detection. Optimization and obfuscator only worsen the problem.

In this chapter, we introduce SLEUTH, a tool that determines whether two functions in a Linux binary are similar. Since it takes binaries as input, SLEUTH does not require source code, from either the alleged victim or the suspect. SLEUTH pipelines semantic and syntactic measures to tackle the binary code theft problem. For this problem, each approach addresses the principal weakness of the other. SLEUTH identifies functions that are simultaneously functionally equivalent and syntactically similar. Functional equivalence abstracts semantic equivalence to the input/output behavior of a function. We compare call graphs to measure syntactic similarity. The syntactic phase follows the semantic and therefore only applies to two functions known to be functionally equivalent. The removal functionally dissimilar functions frees SLEUTH to increase

the sensitivity of its syntactic measure to better filter false positives without the false positive rate it would face comparing arbitrary, unfiltered functions.

Although we focus on a cost-sensitive adversary, for the reasons advanced above, we are aware the binary theft problem is an arms race, one that SLEUTH's advent itself may change. We contend that our approach will change the nature of the race, since it is robust against any semantics-preserving adversary. Our I/O-based semantic heuristic will necessarily equate a victim function and a stolen copy whose semantics are unchanged, while the syntactic measure's similarity threshold can set arbitrarily low and incorporate more syntactic features.

The victim and suspect is selected from among 10 of the most popular C programs on GitHub, a corpus that includes an implementation of Git, a web server, and three databases. When it does not know which compilers and optimizations produced the victim and the suspect, SLEUTH has a mean precision of 88.12%; when obfuscation is used its mean precision is 89.35%; when it knows the compiler and optimization flag, it has a mean F-score of 91.03%.

Our main contributions follow:

(1) We formalize the binary code theft problem as the *hybrid*, pipelined semantic and syntactic comparison of two functions — a victim and a suspect (Section 3.3);

(2) We validate the effectiveness (specificity $\geq 99.94\%$) of approximating semantic equivalence with the output equivalence of functions over random inputs, inspired by the Schwartz-Zippel lemma [74, 94] (Section 3.5.3);

(3) We show that the widely used system call signature heuristic does not effectively identify functions and propose an application function and system call heuristic that does effectively identify functions (Section 3.5.4); and

(4) We realize and evaluate (Section 4.5) SLEUTH, a code theft detector, that detects functions compiled differently from the same source in 10 of the most popular C projects on GitHub in fall 2013 with up 91.03% F-score (Section 3.5.6).

SLEUTH, whose realization is detailed in Section 3.4, is part of the ROSE compiler suite and can be downloaded, along with the scripts and experimental harness used in this work, from `www.rosecompiler.org`.

```
1  void foo(int x) {
2    for (int i = 0; i < 10; i++)
3      x+=i;
4    printf("%i",x);
5  }
```

LISTING 3.1. A simple C function.

```
4005f0: push %rbp
4005f1: mov %rsp,%rbp
4005f4: sub $0x20,%rsp
4005f8: mov %edi,−0x14(%rbp)
4005fb: movl $0x0,−0x4(%rbp)
400602: jmp 40060e
400604: mov −0x4(%rbp),%eax
400607: add %eax,−0x14(%rbp)
40060a: addl $0x1,−0x4(%rbp)
40060e: cmpl $0x9,−0x4(%rbp)
400612: jle 400604
400614: mov −0x14(%rbp),%eax
400617: mov %eax,%esi
400619: mov $0x4006d0,%edi
40061e: mov $0x0,%eax
400623: callq 4004d0 <printf@plt>
400628: leaveq
400629: retq
```

LISTING 3.2. Unoptimized assembly `gcc -O0` produces.

```
400610: lea 0x2d(%rdi),%esi
400613: xor %eax,%eax
400615: mov $0x4006b0,%edi
40061a: jmpq 4004d0 <printf@plt>
40061f: nop
```

LISTING 3.3. Optimized assembly `gcc -O3` produces.

3.2. Illustrative Example

High precision code theft detection in binaries must contend with the difficulty of distinguishing code from data, resolving indirect jumps, and the lack of type annotations, all compounded by differences among compilers and across versions and optimization levels of a single compiler. Indeed, optimizing compilers are effective binary obfuscators. We make the challenge optimizing compilers present to detecting code theft in binaries concrete with an example.

Listing 3.1 contains a simple C program. Compiling Listing 3.1 with `gcc`, using the default `-O0` optimization level, produces the code in Listing 3.2. The mapping from Listing 3.1's source to the assembly in Listing 3.2 is straightforward. Lines 4005f0–4005f4 set up `foo`'s call frame. Line 4005f8 spills `edi`, which

holds x, and 4005fb initializes i to zero. Lines 400604–400607 realize x += i and 40060a increments i. Lines 40060e–400612 implement the loop header; 400614–400623 prepares for and makes the call to printf, while 400628 and 400629 clean up.

Contrast this with Listing 3.3, the assembly gcc produces at -O3, its maximum optimization level. Syntactically, the two are very different. The optimized function is much shorter because the compiler applied loop unrolling followed by constant folding. The compiler hoisted x out of the loop and avoids spilling edi, since only three registers are needed. The compiler also decided that foo does not need a call frame, which eliminates the push, mov, and sub instructions at 4005f0–4005f4 as well as the leaveq in the unoptimized version. Since foo ends with a function call and, at this optimization level, no longer has a call frame, gcc applies tail call optimization and jumps to rather than calls printf.

Detecting code theft in binaries at the granularity of functions requires efficiently and precisely computing the similarity of two binary functions that are in fact closely related, in the face of variety of syntactic differences, like those in Listing 3.2 and Listing 3.3, in the binary encodings of functions at different optimization levels or by different compilers. As the following sections make clear, SLEUTH overcomes these differences by first ignoring them and focusing instead on semantic similarity. Only when it establishes sufficient semantic similarity, does SLEUTH measure syntax similarity, in the form of its abstraction to function call sequences.

3.3. Definitions

Given two programs P and Q with incompatible licenses, our approach to the binary theft problem begins with the insight that, prior to obfuscating countermeasures, binary theft implies that P and Q contain code fragments $p \in P$ and $q \in Q$ that are equivalent *both semantically and syntactically*:

$$[\![p]\!] = [\![q]\!] \wedge [p] = [q], \tag{3.1}$$

where $[\![p]\!] = [\![q]\!]$ denotes semantic equivalence and $[p] = [q]$ denotes syntactic equivalence; in other words, we require both extensional and intentional equivalence.

The literature on code clones studies the syntactic variability of a semantically equivalent pair of code fragments, called clones. Type III clones are code pairs whose syntactic distance, whose exact formulation varies, is less than some bound; Type IV clones are those pairs whose syntactic distance meets or exceeds that bound [20]. Lacking an adversary, syntactic clone tools search for syntactically similar fragments, under

the assumption that, for a fixed platform, sufficient syntactic similarity implies semantic similarity. Thus, these detectors are conservative when they encounter type III clones, where this assumption breaks down.

Our adversary must preserve the semantics of the code he steals; otherwise the theft is pointless. We cannot, however, rely solely on semantic equivalence, since it is prone to false positives when used for theft detection due to innocent, cleanroom implementations of the same specification, where the shared specification might have been imposed by people or by nature, as when writing automotive control software. Unlike traditional clone detection, we cannot rely on syntactic equivalence alone, since our adversary sets that the bound that separates type III from type IV clones, subject to cost constraints on the syntactic transformations he applies. Thus, Equation 3.1 intersects semantic and syntactic equivalence. The guarantee of semantic equivalence allows us to safely equate a much greater syntactic variety, increasing the work the adversary must do to hide their theft. In short, the two forms of equivalence are both needed to tackle the binary code theft problem: syntactic equivalence combats false accusation of theft due to innocent semantic equivalence, while semantic equivalence allows the use of relaxed syntactic equivalence to reduce false negatives, without the danger of identifying behaviorally unrelated clones as suspects.

Of course, Equation 3.1 is too strong, and therefore brittle, in our adversarial setting where we must contend with compiler differences, optimizations, and even obfuscations, both syntactic and semantic. Thus, we relax equivalent to similar in Definition 3.3.1:

DEFINITION 3.3.1 (Replicas). *For the semantic distance function d_{sem} and the syntactic distance function d_{syn}, two code fragments p and q are* replicas *when*

$$r(p,q) \equiv d_{sem}([\![p]\!],[\![q]\!]) \leq \alpha \ \wedge \ d_{syn}([p],[q]) \leq \beta.$$

This definition introduces bounded tolerance for semantic and syntactic variation due to compiler differences, optimizations, and some obfuscations via its two distance functions and their associated thresholds, α and β. We call pairs that are similar under this definition *replicas*, not clones, because the clones have been traditionally defined as either syntactically similar, and thus semantically similar, or solely semantically similar, but not *both*. Our replicas are simultaneously semantic and syntactic clones.

Now we define code theft and code theft mining.

$$
\begin{aligned}
F_i &\triangleq \text{the set of internal functions} \\
F_e &\triangleq \text{the set of } \textit{globally} \text{ external functions} \\
F_e \cap F_i &= \emptyset \\
F = F_e \cup F_i &\text{ is the set of functions} \\
S \subseteq F_i &\triangleq \text{the set of stub functions}
\end{aligned}
$$

FIGURE 3.1. Sets used in this section; each $s \in S$ marshals its parameters and calls a nonstub function $f \in F \setminus S$ in a link unit external to the link unit of s.

DEFINITION 3.3.2 (Code Theft). *Given two programs P and Q with incompatible licenses,* code theft *occurs when*

$$t(P,Q) \equiv \{(p,q) \mid p \in P \wedge q \in Q \wedge r(p,q)\} \neq \emptyset.$$

DEFINITION 3.3.3 (Code Theft Mining). *Given a potential victim P whose license is incompatible with each program $Q \in U$, a corpus of programs, the* code theft mining problem *is to find those programs in U that contain Stolen code,*

$$S = \{Q \mid t(P,Q)\} \subseteq U.$$

Weyuker considered and dismissed a variant of Equation 3.1 as her fourth desirable property of a complexity metric, because the semantic conjunct is undecidable [89]. Our solution is to use functional equivalence to approximate and abstract semantic equivalence, thus rendering the problem tractable. Next, we explain this approximation, and how we realize d_{sem} and d_{syn}.

3.3.1. Semantic Clone Detection. Figure 3.1 defines the set of functions and its notable subset that we use in the following. Let $\vec{I}$ be a finite subset of a function's domain selected uniformly at random.

DEFINITION 3.3.4 (Functional Equivalence). *Two functions $p,q \in F$ are* functionally equivalent *if two permutations π_i, π_o exist such that*

$$\forall \vec{i} \in \vec{I},\ p(\vec{i}) = \pi_o(q(\pi_i(\vec{i}))).$$

This definition restates Jiang and Su's definition of functional equivalence [47], which was inspired by and conceptually rests on the Schwartz-Zippel theorem [74, 94]. It requires the functions p and q to have identical signatures under permutation and relies on random testing in the selection of $\vec{I}$. Intuitively, it should therefore under-approximate the semantic equivalence of p and q to the extent to which $\vec{I}$ is an

inadequate test suite. However, Section 4.5 provides empirical evidence that, for the problem of testing functional equivalence, this intuition does not hold, that $\vec{I}$ is, in fact, adequate. The reason is a conceptual mapping to code of Schwartz-Zippel theorem, which gives a small bound on the number of random tests needed to decide, with high probability, whether two polynomials are equivalent. Indeed, Section 4.5 shows a Schwartz-Zippel-like phenomenon is operative in our data set: our similarity tool is a high precision binary classifier that only needs a few tests to determine equivalence, on average.

Both compilers, when optimizing away constants, and obfuscators can change signatures, so we relax Definition 3.3.4 to tolerate a bounded difference in the signatures of two functions. For the two functions p and q, we first employ two distance functions e_i and e_o to relate their inputs and outputs:

$$\begin{aligned} \forall \vec{i} \in \vec{I},\ \Gamma = \{(\vec{i}, \hat{i}, d_i, \vec{o}, \hat{o}, d_o) \mid \\ (\hat{i}, d_i) = e_i(\vec{i}) \wedge \hat{o} = q(\hat{i}) \\ \wedge (o, d_o) = e_o(\hat{o}) \wedge p(\vec{i}) = o\}, \end{aligned} \tag{3.2}$$

where $d_o + d_i$ is minimal. Equation 3.2 takes the input vector, treat it as a string, and then compute edit distance. Permutation is critical because we treat the input vector as a string. Each tuple in Γ contains the following components: $\vec{i}$ is an input to p; $\hat{i}$ is $\vec{i}$ rewritten to be an input to q; d_i is the distance, or number of edits, between $\vec{i}$ and $\hat{i}$; $\vec{o}$ is p's output on $\vec{i}$; $\hat{o}$ is q's output on $\hat{i}$; and d_o is the distance, or number of edits, between $\vec{o}$ and $\hat{o}$.

We now define two functions that reduce Γ's input and output components to a scalar *similarity score* in $[0..1]$. The function r_i performs this operation over the input components; r_o over the output components.

$$r_i(\Gamma) = \min \left\{ 1 - \frac{d_i}{\max(|\vec{i}|, |\hat{i}|)} \mid (\vec{i}, \hat{i}, d_i, \vec{o}, \hat{o}, d_o) \in \Gamma \right\}$$
$$r_o(\Gamma) = \min \left\{ 1 - \frac{d_o}{\max(|\vec{o}|, |\hat{o}|)} \mid (\vec{i}, \hat{i}, d_i, \vec{o}, \hat{o}, d_o) \in \Gamma \right\}$$

We normalize r_i and r_o to define a universal similarity threshold across all inputs and all function pairs. Without normalization, two inputs can produce arbitrarily different length output vectors, even for a fixed function pair. For example, consider a function that parses a JPEG: if its input is a Word document, it errors; otherwise, it returns a buffer containing the image. Normalization is also essential for reasoning about whether a pair of functions is more similar to each other than some other pair: across all pairs in our corpus, output vectors range from 66% having fewer than 5 values to 21% having more than 10 values. We subtract the raw similarity ratio from one so that 0 is utterly different and 1 is identical. We have left the exploration

of other summarizations than the minimum of these sets, like the average, to future work and, indeed, we expect them to be required as the arms race with our adversary continues.

DEFINITION 3.3.5 (Functional Similarity). *The functions p and q are* functionally similar *when*

$$r_i(\Gamma) \geq t_i \wedge r_o(\Gamma) \geq t_o,$$

where $t_o, t_i \in [0..1]$ are minimal similarity thresholds that convert r_i and r_o into classifiers.

3.3.2. Syntactic Replica Detection. The call graph (CG) of a program is a syntactic structure that arises from a particular decomposition of a computation into functions. It is an effective syntactic signature for a program because changing it, while preserving semantics, is difficult outside of toy languages [70]. Statically extracting a program's exact static call graph is undecidable in general, because the reachability of a given call is undecidable and, even when known to be reachable, dynamic call resolution is expensive [83]. Executing a program on a subset of its input domain traverses paths through a CG that under-approximate the CG. To probe a program's CG, we therefore use dynamic execution against random inputs, as with functional similarity in Definition 3.3.5.

Compilers can change the order and frequency of function calls by inlining, which does preserve semantics. Because inlining can drastically reduce performance due to adverse cache effects, compilers inline very few functions in practice. Thus, most function call paths are similar across different compilations of the same function. We exploit this fact to create a call path similarity measure for two functions. This measure does not rely on function names and therefore works for both stripped and nonstripped binaries.

Our dynamic call similarity measure, defined below in Equation 3.3, compares the edit distance of the call paths of two functions executed over the set of inputs $\vec{I}$. There are three complications. First, to resist inlining, we flatten the paths so they do not encode call level. Second, some of the calls a function makes are to linker stubs inserted by the compiler to resolve the name of a function external to the tested function's link unit, marshal their parameters, and invoke that external function. These stubs spuriously increase the edit distance of two paths, especially when two stubs proxy and invoke the same function. Third, we must rewrite the function names used in the two call paths to exploit the semantic equivalence of functions computed during the semantic phase. For example, if p and q are functionally equivalent, they must be rewritten to the same name, again to reduce spurious edit distance. Next, we discuss each of these complications and our solutions.

Let $f_1 f_2 \cdots f_n$ denote the transitive closure of the calls made during the single-threaded execution of $f(\vec{i})$. The path $f_1 f_2 \cdots f_n$ is flattened; it does not delimit function calls. If $f(\vec{i})$ calls ab where a calls cd and b is a leaf of the call graph and calls no functions, then we write $acdb$. This encoding is blind to the inlining of leaf functions since they dissolve into a sequence of statements that are not recorded, but otherwise resists inlining, compilers frequently employ. For instance, if a were inlined the Levenshtein edit distance between the original $acdb$ and cbd is only 1 not 3 as it would be if we encoded function nesting with $a(cd)b$. To eliminate stubs, we simply drop them from $f(\vec{i})$'s call sequence. This is safe since the external function they proxy must follow them in the transitive call sequence. Finally, we replace each function in the call sequence with its representative element from C, a set of representatives of the functional equivalence class $[F \setminus S]$ computed over all non-stub functions during the semantic phase.

Let $\delta(f(\vec{i})) = f_1 f_2 \cdots f_n$, where $f_i \in C$, denote the transitive closure of the calls $f(\vec{i})$ makes. Recall that C contains no stubs. For the edit distance function e_d, $\forall \vec{i} \in \vec{I}$ defining $\hat{i}$ as in Γ used in Equation 3.2 above, let

$$\Gamma_d = \left\{ 1 - \frac{e_d(\tau_p, \tau_q)}{\max(|\tau_p|, |\tau_q|)} \mid \tau_p = \delta(p(\vec{i})) \wedge \tau_q = \delta(q(\hat{i})) \right\}$$

denote the distance in the paths traversed by p and q over their inputs related by e_i, normalized to fall into $[0..1]$. The dynamic function call distance Δ_d builds Γ_d and then applies r_d to reduce it to a single score in $[0..1]$:

$$\Delta_d(p, q, \vec{I}) = r_d(\Gamma_d) \tag{3.3}$$

where the function r_d could, for example, compute the minimum, average, or maximum value. In the sequel, r_d returns the minimum of Γ_d. To resist function reordering, we experimented with Damerau-Levenshtein as the edit distance, but found that the transpose operator was only used in 0.00005% of the edit operations over our training data set, so Levenshtein was sufficient in practice.

Equation 3.3 is a dynamic under-approximation of the call similarity of two functions, whose quality depends on the degree to which the inputs in $\vec{i}$ exercise the behaviors of the measured functions. Therefore, we balance it with a over-approximate static call similarity measure that computes the multiset Jaccard over the call sites in the two measured functions. We remove backedges from the call graph and define *calls* to return a multiset over F as

$$calls(f) = \{x \mid f \text{ directly calls } x\}$$

and $\overrightarrow{calls}$, removing stubs to improve precision, as

$$\overrightarrow{calls}(f) = calls(f) \setminus S \biguplus_{x \in calls(f)} \overrightarrow{calls}(x).$$

Our static call measure is then

$$\Delta_s(x,y) = \frac{|\overrightarrow{calls}(x) \cap \overrightarrow{calls}(y)|}{|\overrightarrow{calls}(x) \cup \overrightarrow{calls}(y)|}. \tag{3.4}$$

To see how Δ_s works, consider the functions p, which calls aab and q, which calls ade. Let a, c, d be leaf functions, while b calls cd and e calls a. Then we have

$$\Delta_s(p,q) = \frac{|\{a,a,b,c,d\} \cap \{a,a,d,e\}|}{|\{a,a,b,c,d\} \cup \{a,a,d,e\}|} = \frac{|\{a,a,d\}|}{|\{a,a,b,c,d,e\}|} = \frac{1}{2}.$$

Where t_d and t_s are thresholds for these two syntactic measures, we combine them to build a classifier:

$$\Delta(p,q,\vec{I},t_d,t_s) = \Delta_d(p,q,\vec{I}) \geq t_d \ \wedge \ \Delta_s(p,q) \geq t_s. \tag{3.5}$$

We empirically learn effective settings for t_d and t_s from our training set (Section 3.4.5).

Conceptually, the static path measure assists the dynamic measure, which, like any dynamic measure, can suffer from poor path coverage. However, neither measure can distinguish functions that are leaves in the call graph. Both measures trivially deem such functions similar, i.e. functions that make no calls. The usual cause of poor coverage is being unlucky when selecting inputs, especially when approximately an arbitrary probability distribution over the function's input domain with the uniform distribution. In our case, the fact that we select very few inputs, *i.e.* $|\vec{I}| \leq 5$, exacerbates this problem in practice. The reason we select so few inputs is because SLEUTH computes Equation 3.3 along the paths traversed when computing Definition 3.3.5, the functional similarity of two functions under test and this computation requires only a few inputs to classify (Section 3.5.3).

The dynamic measure, in turn, assists the static measure, by reducing its imprecision, such as that caused when obfuscators insert infeasible paths larded with function calls. These unreachable system calls can arbitrarily decrease our static syntactic similarity measure, but, since they are infeasible, have no impact on our dynamic measure. The dynamic heuristic also takes the order of calls into account, which further increases its precision. Thus, we have combined our two measures into Equation 3.5, which logically ands Equation 3.3 and Equation 3.4. When comparing binaries produced by different compilers, we have observed

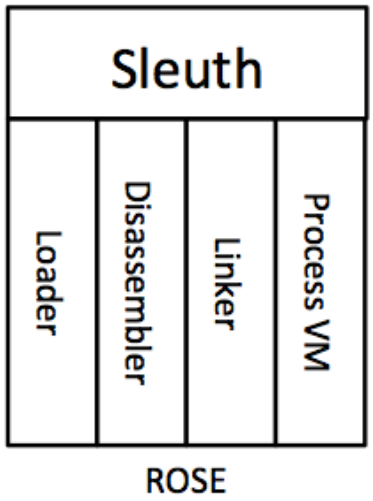

FIGURE 3.2. Architecture of SLEUTH.

Equation 3.5 increases our F-score by 7% on average (Section 3.5.5). We discuss the effectiveness of this heuristic in Section 3.5.4.

3.4. The Realization of SLEUTH

SLEUTH rests on the ROSE compiler infrastructure as shown in Figure 4.6. We first discuss those features of ROSE on which SLEUTH depends. Then we describe how SLEUTH exploits ROSE's extensibility 1) to execute against a lazily instantiated memory when testing the IO equivalence of two functions thereby achieving a practical realization of Definition 3.3.5 and 2) to compute Equation 3.5, which SLEUTH needs to implement its syntactic filter. We close this section by presenting how we validated our implementation of SLEUTH, prior to and independent of Section 4.5, where we compare SLEUTH to related work and evaluate its effectiveness at solving the Code Theft Mining problem, Definition 3.3.3.

SLEUTH compares the output of two functions via testing against the same random inputs. Its key technical innovation is to perform this testing from function, not program, entry. This side-steps the undecidable question of whether a function can be reached as well as the potentially high cost of reaching a function even when a path to its entry is known. Starting from function entry does mean, however, that much of the function's environment is undefined, including the register file, stack frame, and heap. Conceptually, a function's environment, including the IO it performs, is part of its input, which SLEUTH handles by generating randomly for a test run.

In principle, a process VM like SLEUTH can be implemented via rewriting. In practice, however, existing rewriters like Valgrind [61] and Pin [62] have limitations that are not easily circumvented. Unlike Sleuth,

they are tied to specific executable format. They do not provide fine-grained control over linking. Finally, using a process VM automatically isolates the logic and data of the rewriter, such as the code for patching up a function's environment to meet the ABI and to intercept linker invocations, from the subject function's randomized memory and the logic that returns values from the randomized memory if an instruction reads an uninitialized value; rewriters would have to contend with enforcing the needed isolation from scratch.

The* ROSE *Compiler Infrastructure ROSE is a powerful source-to-source compiler infrastructure that includes a loader, a hybrid debugger and process virtual machine [66]. We have published SLEUTH, which is now part of ROSE, under the BSD license at `www.rosecompiler.org`.

3.4.1. Disassembly. Disassembly is difficult because of 1) the interleaving of data and code, 2) computed, indirect jumps, and, in the case of x86 binaries, 3) variable length instructions. To handle these problems and resist obfuscation and anti-disassembly techniques, the ROSE disassembler interacts with the ROSE process VM. Traditional disassemblers, such as IDAPro, rely on compilers to consistently generate code whose patterns can be matched, like recognizable function prologues, and to embed symbols and debugging information. These can be unreliable or absent, especially under optimization. The ROSE disassembler does not rely on this information; it discovers basic blocks then uses its process VM to concretely compute an under-approximation of the targets of indirect jumps and infer the most probable control flow graph over these basic blocks. To side-step the undecidable problem of whether a function is even reachable, not to mention the cost of reaching that function, SLEUTH must be able to execute a function wherever it starts, in order to test the functional similarity of two arbitrary functions, as Definition 3.3.5 requires. Thus, the main task that the ROSE disassembler performs for SLEUTH is the precise identification of function entry. For instance, SLEUTH is aware of and resists the "fake function" call anti-disassembly obfuscation. This technique converts a call into a jump by never returning; evidence of this tactic is when the "fake function" that is the target of the call, pops and discards the return address. Other techniques for finding function boundaries have been explored, some resting on alias analysis and interface identification [17], but were not necessary in the construction of SLEUTH.

3.4.2. The SLEUTH Linker. Linking poses two challenges to SLEUTH. First, compilers are aware of and inline some functions in a language's standard library. Second, as a consequence of its lazily instantiated memory, SLEUTH must contend with address collisions — the inadvertent and incorrect aliasing of pointers — especially in the lower address range $[0..2^8]$ (Section 3.4.3 explains our choice of this interval) from which we uniformly select values and into which the standard linker usually relocates the first library it links.

An example of a compiler inlining standard library functions is GNU's `gcc`. When compiling C programs, `gcc` may (depending on the optimization level) inline some `libc` functions, replacing them with "built-ins". A function whose library calls have been inlined can have arbitrarily different syntactic signatures at different optimization levels under both our dynamic and static function call similarity measures. To our measures, inlined library code is indistinguishable from the lack of the library call. These "missing" library calls cause false negatives when SLEUTH computes the syntactic similarity of two binary encodings of pair of functions or even of a single function.

External functions are function calls made in the code under analysis that are not analyzed; internal function calls are those that are analyzed. To an analysis, external functions are a black box. Typically, external calls are run concretely; SLEUTH follows this convention. System calls are a classic example of external functions. For SLEUTH, any call outside of the analyzed function's compilation unit, notably including calls to the standard library, are external.

Armed with this terminology, we first observe that we cannot identify library calls after they have been inlined. We can, however, *internalize* library calls that may be inlined, transforming them from external to internal functions. So, to solve the problem of inlined library calls, we empirically identified, over our test set (Section 3.4.5) for our set of compilers, those standard library calls that each compiler inlined at least once. For this set of functions, we built a separate library that SLEUTH could analyze. Whenever SLEUTH encounters any of these standard library calls, it continues its analysis of the called library function into our shadow library and does not add the call to the list of calls made by the function under test. Essentially, SLEUTH normalizes the handling of these standard library calls across all optimization levels of a particular compiler.

As discussed in detail below in Section 3.4.3, SLEUTH's memory is lazily and randomly instantiated. When a memory address is read for the first time, SLEUTH fulfills that read with a value selected uniformly at random from $[0..2^8]$. In general, it is undecidable for SLEUTH to know whether the program intends to read an address or non-address data. With this in mind, recall that dynamically linked libraries are relocatable: all of their internal addresses are relative to an offset, which is typically initialized to zero. In conventional execution, the first call to a function in a dynamically loaded library invokes the linker, which resolves all of the library's symbols in the context of the invoking program and using the process' procedure lookup table (accessed via the *@plt functions in ELF) to find a base address for the library that does not conflict with previously linked libraries, then resolves the library's relocatable addresses to concrete addresses in the program's virtual memory.

SLEUTH's address collision problem occurs when it returns a random value in response to a read that, when used as an address, points a memory location that has some other intended purpose, *i.e.* incorrectly aliases the address at which the collision occurs. This in particularly likely to happen with dynamic libraries, because they are relatively large contiguous allocations of memory. When a collision occurs, anything goes — instructions may become data, addresses could be dispatched as instructions, and data can become a jump target — and the test run fails.

SLEUTH's use of randomness means that we cannot prevent address collisions. We do, however, deploy two counter-measures to mitigate the problem. The first is classic and fundamental to probability: multiple trials. The second involves the linker. The SLEUTH simply does not place libraries into $[0..2^8]$, lower address range that SLEUTH uses to fulfill first time reads of a memory location.

3.4.3. Random Lazy Memory. Beginning execution at the entry of an arbitrary function is seemingly pointless. It violates every platform's function calling application binary interface (ABI), not to mention utterly failing to construct the program state the called function expects. Yet this is what SLEUTH does and it successfully tests 66% of all functions. SLEUTH terminates an additional 23% after executing more than 1m instructions and another 3% called the halt instruction or executed an interrupt. To accomplish this feat, SLEUTH uses randomness: it fulfills reads to uninitialized storage locations — heap, stack, and register file — with random values. In other words, reads to previously unread locations are calls to a random number generator. Lazily fulfilling reads is not enough: SLEUTH must initialize some function state, notably globals and the register file to satisfy the ABI.

The data segment, where globals live, is part of every function's context. The loader initializes read-only globals and SLEUTH fulfills reads to these globals normally, returning the value at the named location rather than a random value.

The function ABI requires the construction of a function's call frame, the loading of parameters into the register file, the freeing of other registers for use by callee, under the callee register-saving convention, and finally function clean-up. Thus, SLEUTH initializes EIP (extended instruction pointer) to the address of the first instruction in the tested function, ESP (extended stack pointer) to a random value, chosen to be far away from data and code, and the EBP (extended base pointer), which defines the call frame, to a random value distant, ESP from other allocations. In addition, SLEUTH pushes a nonce into the call frame's return address: when the tested function returns, SLEUTH recognizes the nonce and knows that the function is done.

The callee-saved register convention can hurt SLEUTH's precision by changing how the function consumes random values because the number of registers to save is sensitive to the compiler used and its flags. This

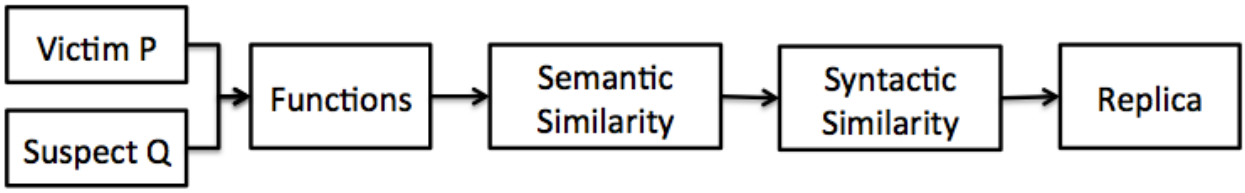

FIGURE 3.3. Overview of SLEUTH replica identification process.

is because callee-saved registers are pushed onto the stack without first initializing them, so each push consumes a random value. SLEUTH obeys the `cdecl` calling convention (*i.e.*, GCC's default for C/C++). By consistently initializing these callee-saved registers, SLEUTH removed a source of false negatives since a random input will then not be consumed for these registers during a read in a test.

Carefully Reusing Randomness SLEUTH seeks to accurately compare the behavior of two functions via random testing. To that end, it must ensure that it runs each functions against the same inputs, in the same order. The challenge here was overcoming instruction reordering: compiler optimizations heavily reorder code, which changes the order of reads, and thus the consumption of random numbers. This fact caused early prototypes of SLEUTH to fail on compilations of the same functions at different optimization levels. To solve these problem, we partitioned our random values into four queues, broadly based on their type which SLEUTH infers from the address range: local, uninitialized globals, unlinked external functions, and register-heap. SLEUTH uses its unlinked external function random pool to assign random addresses to unlinked external functions, essentially naming them for use in both its dynamic and static function call signatures. Partitioning reads in this way allows SLEUTH to achieve its current performance. Conceptually, this partitioning samples the permutation space in Definition 3.3.5 that cannot be exhaustively searched.

While a program is running, SLEUTH lazily records the function's reads and its responses in its memory hash. This hash allows SLEUTH to distinguish initialized from uninitialized memory. For instance, we know that an instruction is reading an uninitialized global when it requests an address in the data segment that is unmapped in the memory hash.

SLEUTH picks uniformly from a user-specified range of values. By default, this range $[0..2^8]$. Following Jiang *et al.*, SLEUTH uses this default because it is the greatest lowerbound of C's primitive types [47]. Thus, this interval has a valid interpretation as each of C's primitive types, *e.g.* char, signed and unsigned short, long, and int, and float, *etc.*, which minimizes the chance of triggering the function under test to immediately abort due to receipt of a value it cannot use. As future work, we place to explore instruction and width sensitive random value generation, as well as exploring distributions other than uniform.

3.4.4. SLEUTH. SLEUTH is a personality of the ROSE process VM, which it extends and customizes. It finds replicas in two phases, as Figure 3.3 shows. It first identifies likely semantic replicas via random testing under the assumption that if two functions are semantically similar their outputs will be similar. For SLEUTH, function output is a value written on the current path. SLEUTH is fundamentally language-agnostic, because it adheres to `cdecl` ABI, but its implementation currently targets C/C++, which do not support out or in-out parameters, which frees us to exclude local variables from consideration as output. Collectively, SLEUTH's lazy Random memory and capability to form the entry point of an arbitrary function allow it to realize Definition 3.3.5.

SLEUTH's random testing of functions is prone to false positives for two reasons: due to the discrete nature of program behavior, as when random testing fails to trigger some behavior that differentiates the two functions, or, in the context of code theft, when two functions are, in fact, semantically similar because they are both legitimate, clean room implementations of the same specification. Thus, SLEUTH computes Equation 3.5 to determine dynamic and static syntactic similarity of two function and further filter the candidates and reduce the false positive rate. At the end of its detection pipeline, SLEUTH produces a set of *replica clusters*, *i.e.*, function pairs that it deems similar.

SLEUTH cannot execute system calls because they are side-effected and external to a process VM. To increase the precision of its syntactic function call measures, however, SLEUTH does want to identify the system call, which it treats as a function. Typically, `libc` proxies system calls. At an `INT 0x80` instruction, the system call number is a constant in the basic block that triggers the interrupt. Thus, to determine syscall numbers for an `INT 0x80` instruction, SLEUTH symbolically executes the basic block, and solves the resulting constraint at the `INT 0x80`.

SLEUTH also supports PE, a.k.a. Windows, executables. Unlike `cdecl` executables where the caller cleans up the stack, PE executables expect the callee to clean the stack. SLEUTH examining the callee's code in the vicinity of its RET instruction and monitors the caller's stack to infer a function's clean-up convention. If cannot infer the convention, SLEUTH assume that the caller cleans up so all we need to do is pop the return address and does not try to pop any arguments.

3.4.5. Validating SLEUTH. SLEUTH required substantial engineering work, since it combines a process VM and the detection pipeline. To validate and guide the development of the process VM, we ran it on the Linux test projects test suite (`ltp.sourceforge.net`). Specifically, we used the following programs from coreutils: `ar`, `basename`, `cal`, `diff`, `patch`, `rmdir`, `tail`, `gzip`, `diff`, `patch`, `grep`, `sed`, and `ls`. We

selected these programs for our training set because they are relatively small, yet important targets that exhibit challenging properties like frequent system calls and heavy reliance on IO.

Over this test suite, we iteratively improved SLEUTH. First, we equipped SLEUTH with ability to marshal data to and from our virtual machine memory to the native memory so that SLEUTH could concretely run programs from entry, allowing us to validate its correctness as process VM. To validate and improve detection, we selected function pairs from it that we knew to be the same because they were compiled from the same source at different compiler settings or which we knew to be different. We then ran SLEUTH on these pairs to identify all the false positives and false negatives, then reran them logging all their inputs and outputs. We laboriously, manually compared each function's logs and their disassembled assembly code to determine where that version of SLEUTH was going wrong. This is how we discovered the need to partition the pools of random values used to fulfill reads by address, like distinguishing locals and globals (Section 3.4.3).

This separation of our data sets into training data, described here, and the test data we use in Section 4.5, is strong evidence that our results generalize, and do not merely overfit our training data.

3.5. Evaluation

Before evaluating SLEUTH's performance at detecting code theft, we first describe our evaluation environment and our test corpus. Section 3.5.1 describes our corpus, its selection, and processing. Section 3.5.2 describes our experimental methodology. Section 3.5.3 determines the setting for the number of runs against random test inputs, demonstrating the surprising specificity of our semantic heuristic; Section 3.5.4 justifies our syntactic measures, notably showing that function call sequences must incorporate application functions and cannot rely solely on system calls in our setting. Section 3.4 details the realization of SLEUTH used in this evaluation. We describe the search for effective settings for SLEUTH, over our training set, in Section 3.5.5. Having found the settings, Section 3.5.6 then evaluates SLEUTH's ability to precisely and accurately detect code theft. We close with a discussion of the limitations of the current implementation of SLEUTH, in Section 3.5.7.

We performed the experiments described in this section on an AMD Opteron(tm) Processor 6380 2.5Ghz 32 cores with 128 GB RAM and a 1TB hard drive. SLEUTH's semantic analysis phase takes 6 hours 35 minutes to run semantic analysis over our test set. Using results from the semantic analysis, the syntactic analysis takes 5 hours.

3.5.1. Corpus. We collected the top 10 most popular C programs on GitHub on September 28th 2013 that compiled to a single executable or library. These programs are libgit2, libharu, libwebsockets, libwhitedb,

nginx, redis, slash, sparkey, sundown, and yajl. They include an implementation of Git, a web server, and three databases.

Like many classifiers, SLEUTH requires sufficient data. Its syntactic phase simply cannot reliably classify small functions, which make few function calls. Thus, in line with Chaki *et al.*'s curation [21], after computing the functional equivalence of and partitioning all our functions, we discard the partition E and its functions if it satisfies $\text{argmax}_{f \in E} \max(|f|) < 100$. The cutoff of 100 was chosen empirically: the proportion of leaf functions, which are all similar under our syntactic heuristic, drops from 14% of all functions to 4% for functions with 100 or more instructions, as detailed in Section 3.5.4. Our test set contains 17,520 functions, 7594 after filtering out functions with fewer than 100 instructions.

3.5.2. Experimental Setup. SLEUTH does not require source. This allows it to search for software theft without contending with the combinatoric explosion of binary variants compilers and their flags can generate from a single source file. In these experiments, however, we need a ground truth in order to measure SLEUTH's performance. Thus, we restricted our test set to programs for which the source is available and coped with the combinatorial explosion by restricting our study to three compilers and their optimization flags.

Given the pair of functions v and s, SLEUTH executes the pair k times over identical, randomly generated inputs, generating three sets of measures, one set derived from each of the measures in Section 3.3. Runs in which either v or s do not terminate or error are ignored.

Tolerating differences in input is expensive both in computation and storage, so, in this study, we fix r_i to be the constant function 1 in Definition 3.3.5 and do not tolerate differences in two functions' input vectors. Instead, we focus solely on output, use multiset Jaccard, which is order insensitive and inexpensive to compute, and, in Definition 3.3.5, set

$$r_o(\Gamma) = \min \left\{ \mathit{Jaccard}(\vec{o}', \hat{o}') \mid (\vec{i}, \hat{i}, d_i, \vec{o}, \hat{o}, d_o) \in \Gamma \right\}$$

where $'$ converts a vector into a set of its components: $v^{n\prime} = \{x \mid x = y_i \wedge (y_1, y_2, \cdots, y_n) = v^n\}$.

To order function pairs by similarity, we combine each of these three measures with thresholds to convert SLEUTH into a classifier. Let $\vec{T} = [0..1]$ (minimum I/O similarity) $\times [0..1]$ (minimum dynamic function call similarity) $\times [0..1]$ (minimum static function call similarity) denote SLEUTH's setting vector.

Let $C = \{\texttt{gcc}, \texttt{icc}, \texttt{llvm}\}$ be the set of compilers and $X = \{\texttt{O0}, \texttt{O1}, \texttt{O2}, \texttt{O3}, \texttt{Os}\}$ be the set of optimization flags. To instantiate the Code Mining Theft problem (Definition 3.3.3), the experiments that follow pick P from among our benchmarks, compilers $c_v, c_s \in C$, and flags $x_l, x_j \in X$, then compile P twice, producing a

victim $V = c_v(x_k, P)$ and a suspect $S = c_s(x_l, P)$. In so doing, we are imagining P is incompatible with its own license. We sample C and X, with replacement, allowing $c_s = c_v$ and $x_l = x_j$, because the same compiler and optimization flag may generate both the victim and suspect. This experiment is equivalent to an actual theft, while avoiding the engineering difficulty of automatically transplanting a stolen function. Since we create both V and S from P, we know which functions should match and which should not despite differences in the compiler and the optimizations applied. Let $sleuth(k, \vec{t}, v, s)$ denote SLEUTH asserting $v = s$, using k runs over random inputs with the settings vector $\vec{t}$. To measure SLEUTH's accuracy, we compute

$$\begin{aligned} &Sleuth(k, \vec{t}, C, X, P) = \\ &\{(v, s) \mid sleuth(k, \vec{t}, v, s) \wedge v \in c_v(x_k, P) \wedge s \in c_s(x_l, P)\}, \end{aligned} \tag{3.6}$$

then check, for each (v, s) in this set, if $v = s$ actually holds. The set *Sleuth* contains SLEUTH's positive results over P at the given settings; its negative results are therefore $P \times P \setminus Sleuth$.

Our test set contains 7594 functions after filtering out functions with fewer than 100 instructions (Section 3.5.1) and 5543 after filtering those the semantic phase groups into large clusters (Section 3.5.5.2), far too many for us to manually determine their pairwise similarity. Fortunately, our experimental design gives us the ground truth: since we produce both the suspect and the victim binaries from the same program, we simply do not change a "stolen" function's name before creating the victim. Thus, we know that two functions with the same name in the victim and the suspect are the same. This allows us to detect both false positives and false negatives. As a sanity check, we manually inspected 30 function pairs selected uniformly at random: in every case, identical function names coincided with semantic equivalence.

After filtering, our corpus has 5543 functions and 10 programs, so for each experimental run over a program, SLEUTH makes approximately $(\frac{5543}{10})^2 = 306916$ comparisons on average. Although Equation 3.6 takes $O(n^2)$ comparisons, it is embarrassingly parallelizable since each function test and each pairwise similarity calculation are mutually independent. We exploit this parallelism. For instance, we concurrently test all the functions in a file to save disassembly. We also split the computation into intercommunicating phases. Our computation runs in phases — generate random inputs, test functions, perform the semantic comparison, and perform the syntactic comparisons — each of which is internally concurrent. These phases intercommunicate using a database as a blackboard: this facilitated troubleshooting, persisted our results, and allowed to pose and answer unanticipated queries without re-running the experiment.

The database that stores our results exceeds 1TB, which we do not have the resources to host longterm. SLEUTH is open source, however, as are the scripts we used to conduct these experiments. These scripts

download our data sets, so interested parties can download everything needed replicate our results from www.rosecompiler.org.

3.5.3. Random Testing as a Classifier. Inspired by the Schwartz-Zippel lemma, Definition 3.3.5 is a binary classifier that requires only a *few* random tests to semantically decide whether two binary functions are similar or dissimilar. The governing intuition that underlies Definition 3.3.5 is that two functions, executing over a small set of identical random inputs, will generate output whose similarity closely reflects the semantic similarity of the two functions: two semantically different functions will, in general, rapidly exhibit different behavior over uniformly chosen identical inputs.

Among a population of functions, most functions are dissimilar, because similar functions tend to be reused within a project and small differences in how to modularize and decompose a problem into functions compound across projects. Thus, we expect the majority of SLEUTH tests to be negative. In a clinical setting, specificity [54, 79] measures the ability of a test to correctly identify healthy patients; in our setting, we use it to measure the ability of SLEUTH to correctly identify dissimilar functions. In medicine, specificity is extremely important: you do not want healthy people to falsely think they have HIV. For the code mining problem, specificity means that we will not waste resources suspecting two functions are similar when they are, in fact, dissimilar.

Flipping a fair coin as a classifier has 50% specificity. The experiment that follows show that our random testing classifier has a recall of 80% and specificity of 99.96% with a standard deviation of 0.02%. In a large population of mostly negatives, we almost perfectly classify negatives as negatives, while correctly classifying most positives as positives. This extremely high specificity compares favorably with the current HIV/AIDS test where a large study reported specificities of 99.8% and greater than 99.99% [23], which is unusually high for clinical tests [54].

It is not obvious that a Schwartz-Zippel like phenomenon holds over binaries. First, it would seem that two arbitrary functions, even executed over identical input, could very well generate output whose similarity fell anywhere in the interval $[0..1]$, although skewed toward 0%. An extreme example of this phenomenon is a pair of boolean functions that each return true only at one point in their domains and do not coincide on that point. Indeed, this accounts for many of the false positives of our approach. Second, beginning execution from function start means that much of a function's environment is missing and becomes input. SLEUTH generates this input through the mechanism of its lazily, randomly instantiated memory (Section 3.4.3). The resulting random memory will violate the function ABI and trigger infeasible paths. Executed against this

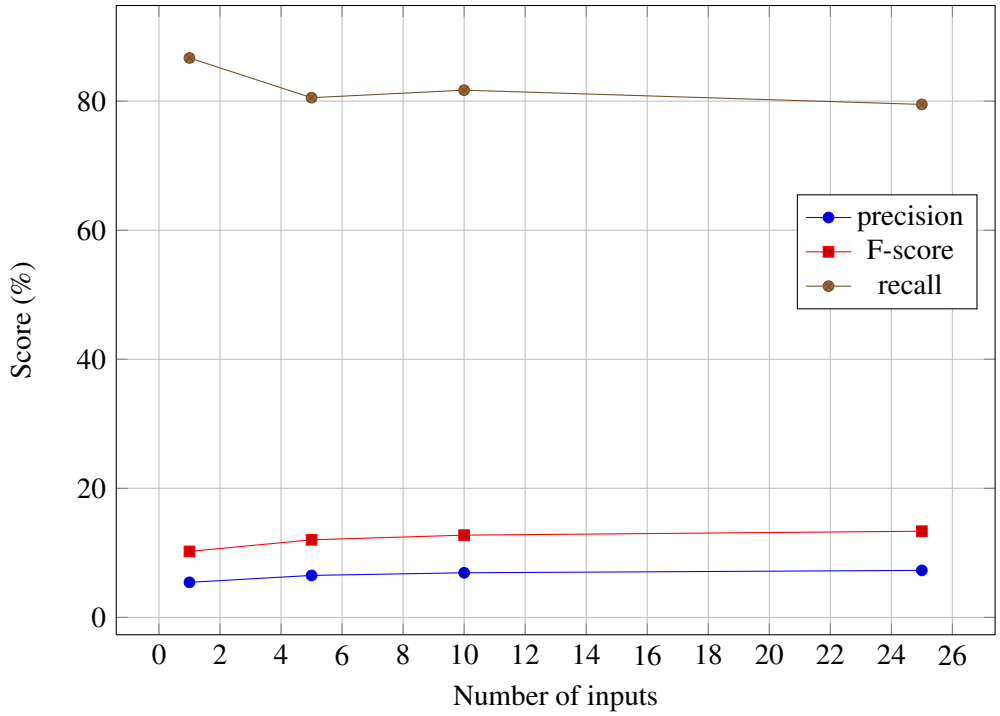

FIGURE 3.4. As a binary classifier using I/O equivalence, SLEUTH is stable, albeit not very precise before its syntactic filtering, as x, the number of inputs increases in $\mathit{Sleuth}(x, \langle 1, -, - \rangle, C = \{\texttt{gcc}\}, X, P)$, so we set k, the number of SLEUTH's test runs, to 5 in the rest of our experiments.

memory, functions will obey the dictum "garbage in, garbage out" and generate output that, intuitively, could differentiate even semantically similar functions.

To validate Definition 3.3.5 effectiveness as a classifier, we constructed an experiment using 5 (the next experiment makes this choice clear) inputs with $C = \{\texttt{gcc}\}$ and $X = \{\texttt{O0}, \texttt{O1}, \texttt{O2}, \texttt{O3}, \texttt{Os}\}$ in Equation 3.6 that counts what percent of function pairs in Definition 3.3.5 are 0% to 100% similar in 5% increments. If our semantic clone detection is a perfect binary classifier, then each pair should either be 0% or 100% similar. The result showed that 4.53% of the functions are 100%, while 87.18% are 0% similar. In our experiment, only 0.01% of all functions are $\geq 50\%$ and $< 100\%$ similar and 94.09% of all functions are $\leq 15\%$ similar. This shows that our approach is a binary classifier since almost no function pairs are partially, or near, similar.

We have just shown that the semantic component in Definition 3.3.5 is a binary classifier, but how stable and accurate are its results? To find out, we aggregate scores for semantic similarity by testing each pair against 1, 5, 10, and 25 random inputs with $C = \{\texttt{gcc}\}$ and $X = \{\texttt{O0}, \texttt{O1}, \texttt{O2}, \texttt{O3}, \texttt{Os}\}$ in Equation 3.6. In this experiment, we used the Jaccard index to measure output similarity. Figure 3.4 demonstrates that Definition 3.3.5's results are quite stable after 5 inputs. Thus, we set SLEUTH's output similarity threshold to

Program	All		≥ 100	
	Static	Dynamic	Static	Dynamic
libgit2	63.06	16.53	77.09	12.56
libharu	2.72	1.63	4.82	1.20
libwebsockets	51.25	30.00	72.73	39.39
libwhitedb	74.32	27.57	85.71	40.82
nginx	67.44	7.79	82.51	7.61
redis	46.68	8.57	68.17	16.54
slash	47.83	4.23	57.72	4.03
sparkey	31.11	17.78	31.03	17.24
sundown	22.13	9.84	28.21	5.13
yajl	29.48	14.08	12.50	8.33
Total	23.33	5.93	40.0	15.13

TABLE 3.1. Percentage of functions that call functions that may (static) or do (dynamic) make system calls; there are 17520 functions in total of which 7594 have 100 or more instructions.

1 in the sequel. Figure 3.4 also demonstrates the high recall of SLEUTH's semantic phase and its principle shortcoming: false positives as manifest in its precision, dragging down its F-score. This imprecision has two main sources: 1) independent implementation of the same specification or 2) the existence of a class of functions with different rare behaviors that uniformly random testing is unlikely to trigger. To combat imprecision, we employ our syntactic measures. Next, we discuss why we did not reuse an existing syntactic measure.

3.5.4. The Raison d'Être of our Syntactic Measures. Our thesis is that we must hybridize semantic and syntactic similarity measures in sequence to detect code theft. The semantic phase discovers behavioral similarity independent of optimization and syntactic changes; the syntactic combats the semantic component's tendency toward false positives, as we just made concrete in Section 3.5.3. We could have used an existing syntactic heuristic. Why did we invent two new ones in Section 3.3.2 that we combine in Equation 3.5? The simple reason is that none is is well-suited for detecting code theft (Definition 3.3.2) in our adversarial setting. We validate this claim against three prominent syntactic similarity heuristics for binaries: the popular system call signature [87], *binary clone detection* [69] and *normalized compression distance* [39].

Our syntactic heuristic builds signatures from function calls as well as system calls. Previous work has focused on system call signatures since they are immutable to most adversaries and effectively detect whole program copies [88] or malware [40]. They work well for programs with a rich system-call behavior. Many complex functions, however, including some that take man-decades of specialized labour to produce, *e.g.* video (de-)compression in ffmpeg, do not have rich system call behavior. Our concern is the theft of functions,

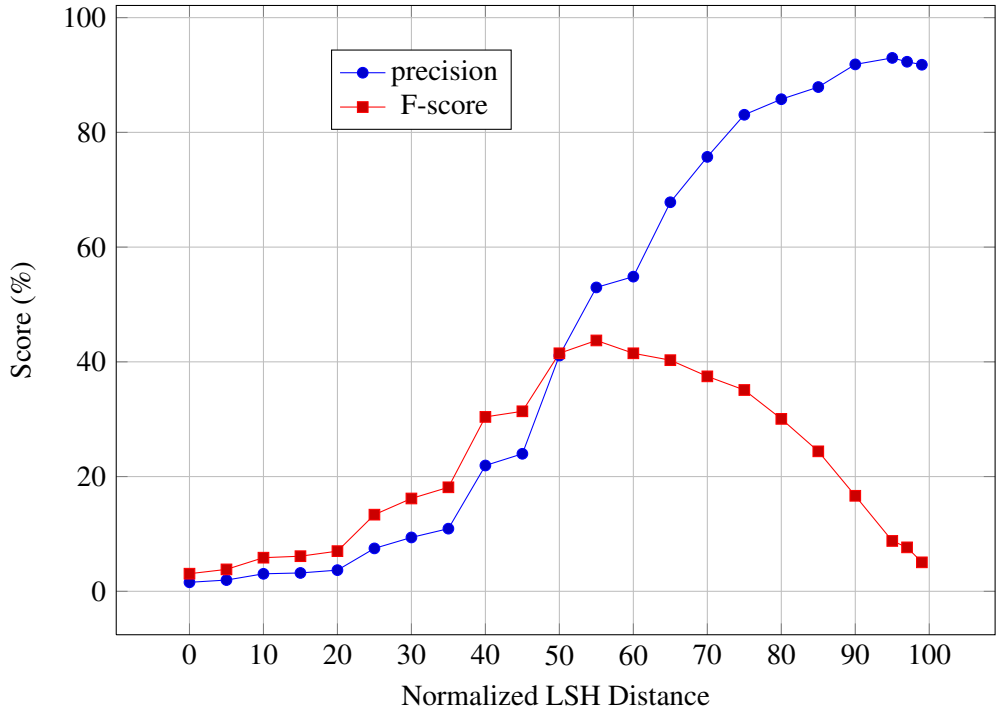

FIGURE 3.5. Effect of minimal allowed output similarity on precision and F-score detecting clones using Sæbjørnsen *et al.* [69] feature vectors.

and, in this setting, system call signatures are imprecise. To demonstrate this, we used SLEUTH to extract a CFG for each function and to count how many system calls functions make. We harvested system calls from this CFG, identifying these calls as described in Section 3.4.4. For a given function, we counted its direct system calls, those calls it makes in its body, and its indirect system calls made in the transitive closure of the functions it calls, both dynamically along paths, and statically across all paths.

Table 3.1 shows that most functions do *not* make system calls, directly or indirectly: 76.67% of all or 60% with 100 instructions or more. Further, many system calls are conditioned on input as our dynamic results demonstate: only 5.93% of all functions or 15.13% of functions with 100 instructions or more make a system call. Thus, system call signatures simply cannot identity majority of functions in our test set.

In previous work, we defined and implemented binary clone detection based on syntactic feature extraction [69] and applied it to the detect of code theft. Related work, like *e.g.* Chaki*et al.* [21], utilize feature extraction as well. The binary clone detection algorithm slides a window over a binary, normalizing the registers and memory locations within the window, then clusters the normalized binary snippets with LSH [29] to discover clones. We did not design this algorithm to handle compiler optimizations, let alone obfuscations, but it remains the state of the art of binary clone detection.

Figure 3.5 shows the precision and F-score of the binary clone detection algorithm for different minimum output similarity thresholds with compiler optimization as an adversary. Binary clone detection achieves a 42% F-score when outputs were required to be at least 50%. We were unable to find a minimal output similarity setting in our training set for which the binary clone detection algorithm effectively filters the semantic heuristic to achieve high precision without accepting an F-score of 42% or less and an even lower precision.

Hemel *et al.* applied normalized compression distance (NCD) to the problem of detecting whole program code theft [39]. They reported impressive results — 91% precision and 72% recall — in their problem setting, for their NCD, Reuse-C and Reuse-D measures. To evaluate their measures in our setting, we replaced SLEUTH's functional similarity measurement pipeline with each of Hemel *et al.*'s measures in turn. None of Hemel *et al.*'s measures performed well in our setting. Their best F-score is 0.7%. This may be due to the fact that functions are generally much smaller than programs and therefore do not provide compressors many opportunities to exploit common subsequences. Another cause may be the fact that our experiment tests many more victim/suspect pairs than do Hemel *et al.* — 7594 vs. 10.

3.5.5. Tuning SLEUTH. SLEUTH has three settings: output similarity of functional equivalence testing, dynamic function call similarity, and static function call similarity. Together, these form SLEUTH's setting vector, whose settings we now explore over our training set. Large clusters of putatively similar functions dominate SLEUTH's false positives. We explore this phenomenon, then and the effects of filtering these clusters to find a maximum cluster size to analyze that increases SLEUTH's precision and F-score, while retaining the majority of our victim/suspect pairs.

3.5.5.1. *Setting Thresholds.* Section 3.5.3 shows that SLEUTH's semantic similarity under Jaccard distance is an effective binary classifier that assigns similarity other than 0 or 1 to less than 8% of all function pairs in our test set. Furthermore, only 0.01% of those 8% have a similarity greater than 50%. Thus, we set SLEUTH's output similarity threshold for its functional equivalence via fuzz testing to 1. Two thresholds remain: SLEUTH's dynamic and static path thresholds.

Figure 3.6 demonstrates the performance of SLEUTH's dynamic function call heuristic as a function of the minimum normalized edit distance between the function call paths of a potential victim are compared against those of a suspect. The minimum normalized edit distance considers a victim and a suspect similar if their least similar (minimum) pair of function call paths is at least x% similar. In this experiment, we fixed C to `gcc` while varying the optimization level. Each point is therefore the mean of $7594 \cdot 15$ (all pairs chosen from X) victim/suspect pairs. Figure 3.7 shows the dynamic function call heuristic's performance when the

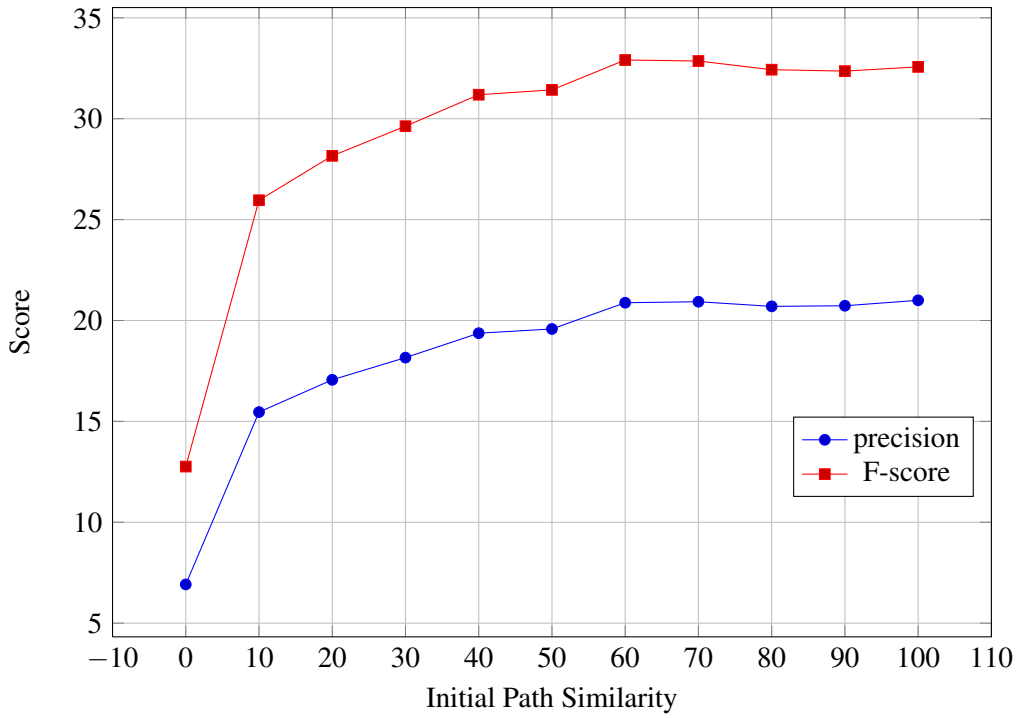

FIGURE 3.6. Dynamic function call measure evaluated over $\mathit{Sleuth}(5, \langle 1, x, 0\% \rangle, \{\texttt{gcc}\}, X, P)$ in Equation 3.6; at $x = 40\%$, the minimal initial similarity of k paths through two functions was at least 40%, the paths could be rewritten to be equivalent, and SLEUTH's precision was 19.37% and its F-score 31.19%.

optimization level is fixed to `O3` ($X = \texttt{O3}$), while the compilers vary. Here each point is the mean of $7594 \cdot 6$ (all pairs chosen from C with replacement).

These figures understate SLEUTH's performance. We computed the standard deviation of each point. In $C|\texttt{gcc}$ in Figure 3.6, the standard deviation of the F-score was 8.59% and that of precision was 5.65%. In $X|\texttt{O3}$ in Figure 3.7, the standard deviation of the F-score was 3.25% and that of precision was 2.49%. These low standard deviations mean that the scores cluster around their means and, therefore, that SLEUTH's performance is quite stable across all of the experimental settings. From these results, we infer that an effective normalized function call similarity setting is 60%.

SLEUTH's dynamic function call heuristic depends on its semantic, functional equivalence, heuristic in two ways. First, it requires the computation of functional equivalence classes using Definition 3.3.5, so that all of these functions can be normalized to a single name for matching in the syntactic phase. Second, the dynamic path function call signatures are collected along the paths executed during the functional equivalence testing. Since SLEUTH's functional testing determines equivalence in a few runs, our dynamic function call heuristic's coverage is low. Thus, we complement it with our static function call heuristic.

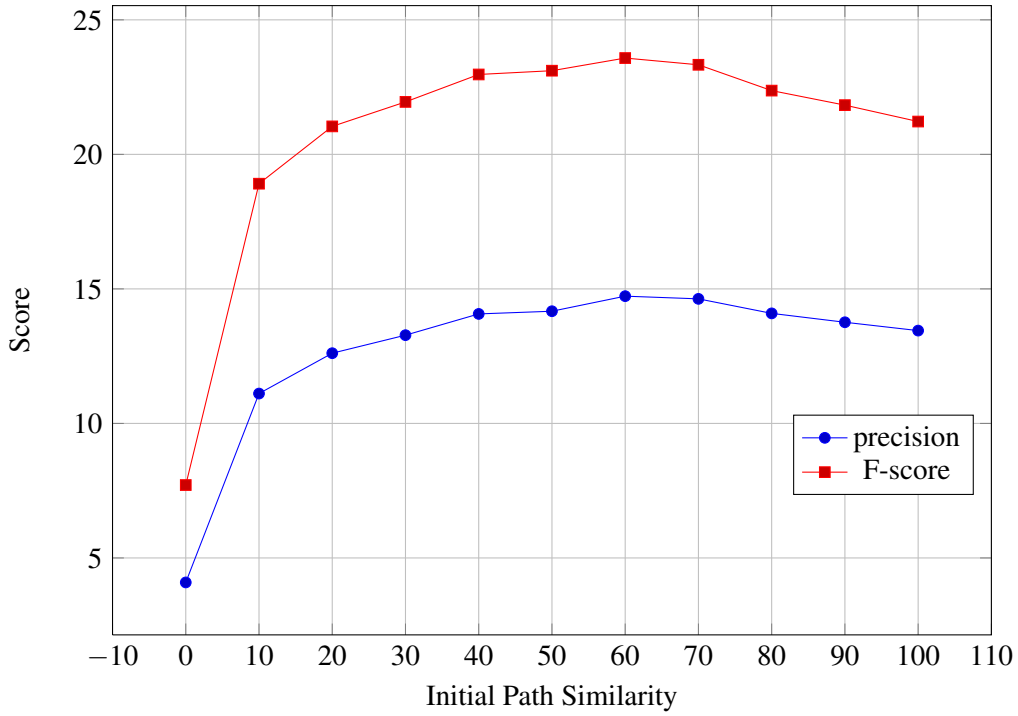

FIGURE 3.7. Dynamic function call measure evaluated with $Sleuth(5, \langle 1, x, 0\% \rangle, C, X, P)$ where different compilers produce the victim and the suspect ($c_v \neq c_s$) in Equation 3.6; Figure 3.6's describes how to read a point.

Figure 3.8 repeats the experiment depicted in Figure 3.6, but with our multiset Jaccard measure over all calls made in a victim and a suspect as the independent variable. Here, a particular value of x means that all victim/suspect pairs have a Jaccard measure greater than or equal to x. At $x = 40$, the mean of all of the pairs whose multiset Jaccard is at least 40 slightly exceeds 40. SLEUTH's static function call measure has superior precision and F-score to the results for binary clone detection in Figure 3.5. Figure 3.9 repeats the Figure 3.7 experiment with multiset Jaccard as the independent variable over all calls that statically appear each victim and suspect. Here, SLEUTH exhibits high precision but benefits from knowing the compiler, as the fall in F-score between Figure 3.8 and Figure 3.9 makes clear.

Alone, these figures are inconclusive: Figure 3.8 suggests setting the similarity to 100% and requiring a perfect match; Figure 3.9 maximizes both precision and F-score at 60%, but suggests 95% is best if one's goal is to maximize precision. To gain more insight, we turn to standard deviation. In Figure 3.8, the standard deviation of F-score is 23.81% and that of precision is 7.53%; in Figure 3.9, the standard deviation of F-score is 2.31%, and that of precision is 5.5% standard deviation in precision. These low standard deviations demonstrate the stability of SLEUTH's static function call signature. Eyeballing both figures, we choose 70.

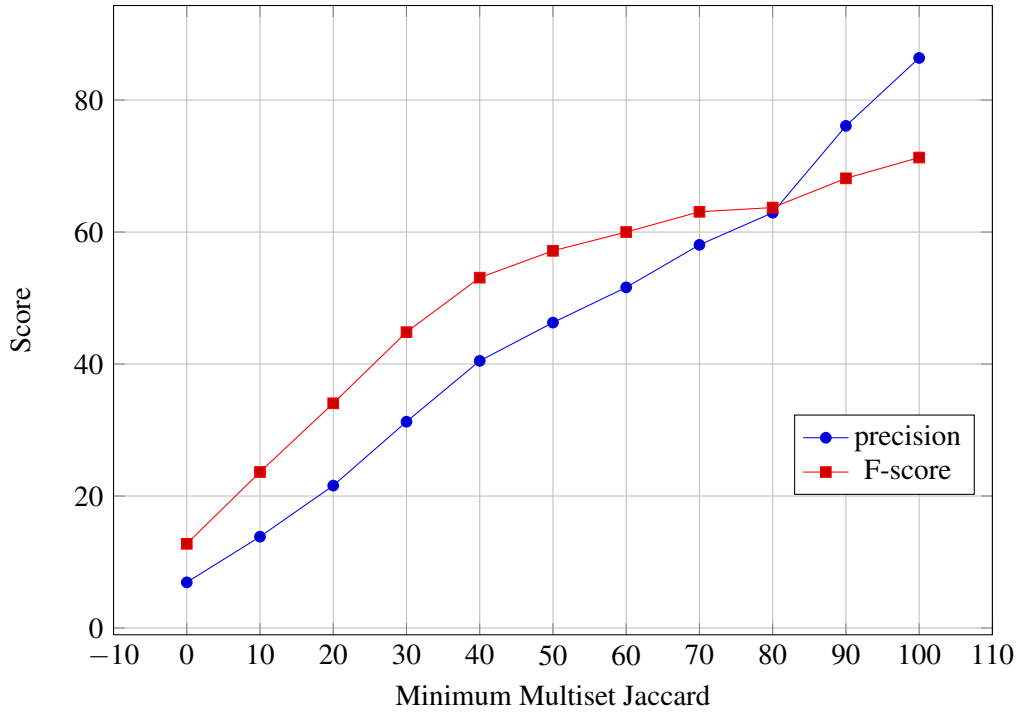

FIGURE 3.8. Static function call measure evaluated with $\mathit{Sleuth}(5, \langle 1, 0\%, x\rangle, \{\texttt{gcc}\}, X, P)$ in Equation 3.6.

To test whether our settings vector $\langle 1, 60\%, 70\%\rangle$ is difficult to improve upon, we looked for counter-examples among 30 setting vectors, chosen uniformly at random. The results were unsurprising. They confirmed our intuition that higher settings trade recall for precision and degrade F-score. Most to the point, we found no setting that improved on $\langle 1, 60\%, 70\%\rangle$.

3.5.5.2. *Filtering Clusters.* Under Definition 3.3.3, most functions should differ from most other function because they solve different problems or have different, unrelated implementations. Thus, most function should be similar to only a few other functions. Empirically, however, some classes of functions violate this rule and contribute disproportionally to Equation 3.5's false positives. These functions tend to make few function calls or exhibit low IO entropy, *e.g.* almost always return true (or false).

We hypothesized that large clusters of similar functions have two causes: 1) cleanroom implementations of commonly occurring specifications, like sorts and trees, or 2) accessors which tend to be indistinguishable to SLEUTH, since they have similar read patterns and SLEUTH's lazily instantiated memory therefore ensures their reads return the same values. To validate this hypothesis, we picked 30 random theft candidate pairs from the union of all clusters with more than 100 elements. 28 of these pairs were accessors and two were wrappers around internal interfaces that were detected as clones of the interfaces they wrap.

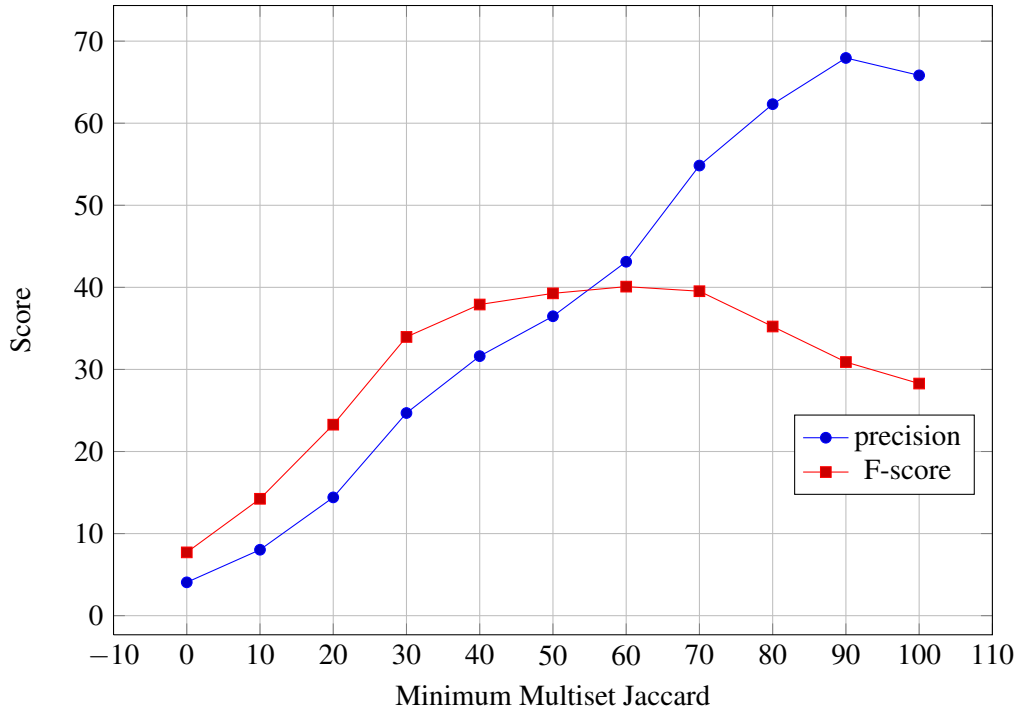

FIGURE 3.9. Static function call measure evaluated with $\mathit{Sleuth}(5, \langle 1, 0\%, x\rangle, C, X, P)$ where different compilers produce the victim and the suspect ($c_v \neq c_s$) in Equation 3.6.

This result suggested that filtering large clusters would expediently improve SLEUTH's performance. To see the impact of this filtering on SLEUTH's performance, we conducted two experiments. The first experiment uses the settings $\langle 1, 60\%, 70\%\rangle$; Figure 3.10 shows its results. The second experiment uses the settings $\langle 1, 0, 0\rangle$; Figure 3.11 shows its results. In both figures, we see SLEUTH's precision climb, from right to left, as we prune more and more clusters. At 10^1, in both figures, F-score begins to fall dragged down by recall, because, at this point, we are throwing away 27% of all of our pairs. The distribution of percent of function pairs per cluster size is a power law: singletons clusters comprise 46.9% of all the function pairs and, when we throw out clusters larger than 80, we lose only 10% of the function pairs.

As these results make clear, SLEUTH's performance greatly improves when large clusters are filtered, without losing a significant portion of the training set's victim/suspect pairs. For these reasons, we report results in Section 3.5.6 after filtering clusters larger than 10, which improves both our precision and F-score, while retaining 73% of our pairs.

3.5.6. SLEUTH vs. Its Adversaries. SLEUTH requires selection and tuning of its parameters. Section 3.5.5 describes how we set its parameters. Armed with effective settings, we evaluate SLEUTH's performance against its menagerie of adversaries. We measure SLEUTH's performance in terms of precision,

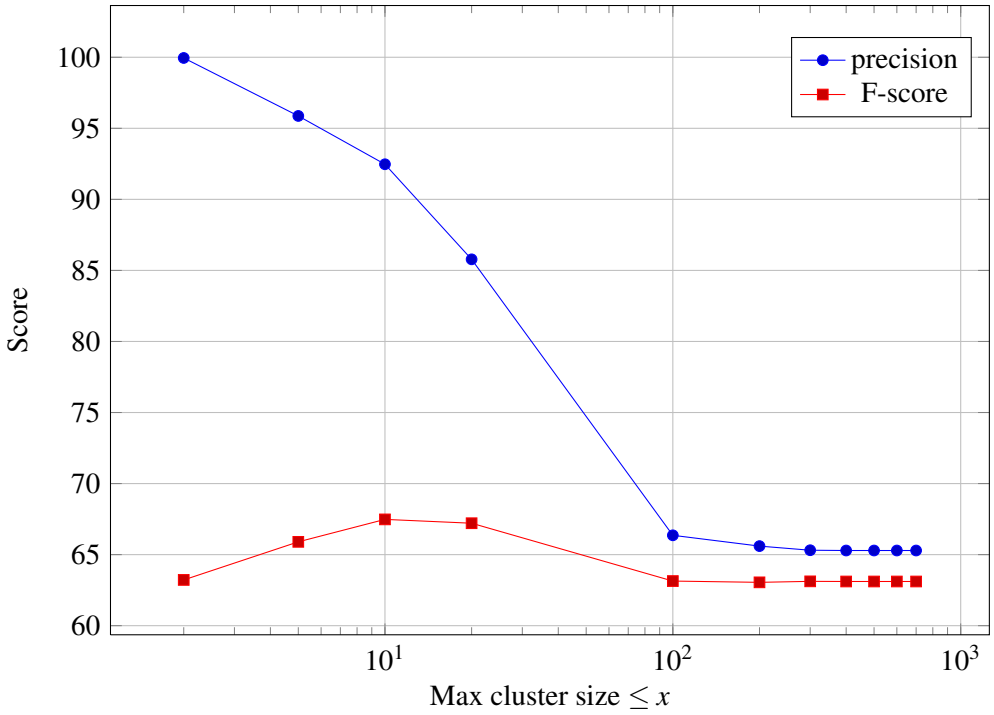

FIGURE 3.10. Filtering victim/suspect clusters by size under *Sleuth*$(5, \langle 1, 60\%, 70\% \rangle, \{\texttt{gcc}\}, X, P)$ in Equation 3.6.

which must be high to avoid false accusations of theft, and F-score, to show it remains sensitive to theft despite its high precision.

3.5.6.1. Optimizer Resilience. First, we ask "Is SLEUTH's performance resilient against compiler optimizations?". This experiment demonstrates SLEUTH's performance when we know the compiler both the victim and the thief used. To answer this question, we fix the compiler to `gcc`, the most popular Linux compiler, while varying the optimization flags. To this end, we compute Equation 3.6 with *Sleuth*$(5, \langle 1, 60\%, 70\% \rangle, \{\texttt{gcc}\}, X, P)$. For all 15 (5 choose 2 with replacement) pairs of optimizations, we compiled each program in our corpus to produce 10 pairs of victim V and suspect S binaries. For each of these (V, S) pairs, we compared each function in V against all the functions in S, as described in Section 3.5.2. A comparison is true if the result is in agreement with the ground truth or false if it is in disagreement. A data point in the box plot is the F-score or precision over all comparisions of a function in V against a function in S.

Figure 3.12 shows the results. The box plots capture the results of comparing each $v \in V$ to each $s \in S$. SLEUTH does particularly well across the same optimization level, as the boxplots for $(\texttt{O0}, \texttt{O0})$, $(\texttt{O1}, \texttt{O1})$, $(\texttt{O2}, \texttt{O2})$, and $(\texttt{O3}, \texttt{O3})$ demonstrate. $(\texttt{O2}, \texttt{O0})$ and $(\texttt{O2}, \texttt{Os})$ have a lower F-score due to imprecision in the handling of pointers in the current implementation of our lazily instantiated memory, for details refer to the

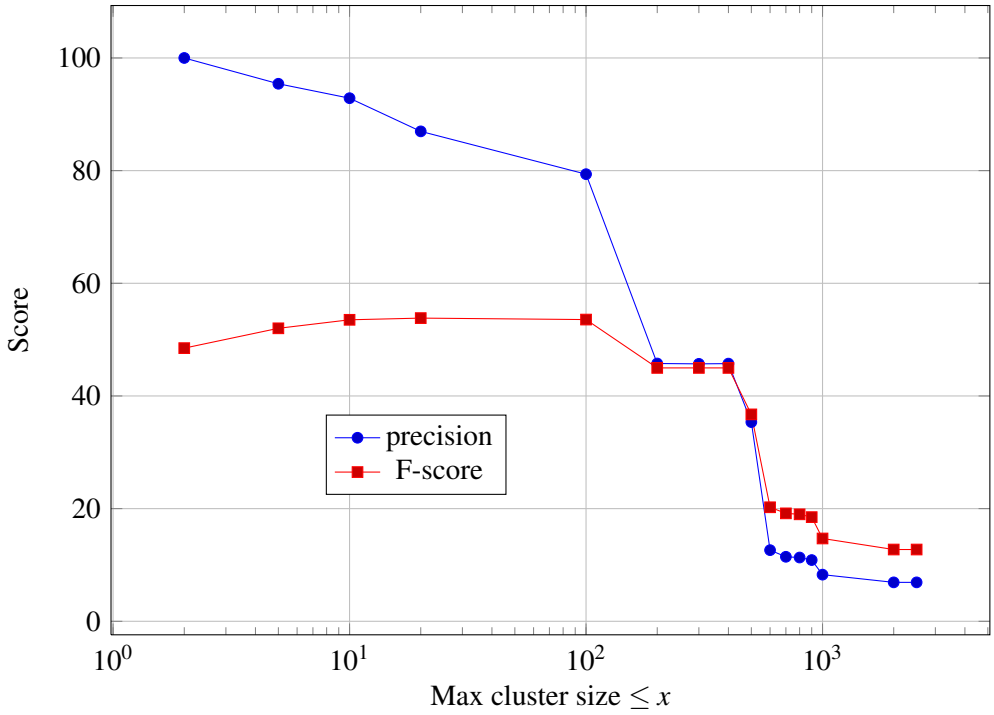

FIGURE 3.11. Filtering victim/suspect clusters by size with *Sleuth*$(5, \langle 1, 60\%, 70\% \rangle, \{\texttt{gcc}\}, X, P)$ in Equation 3.6.

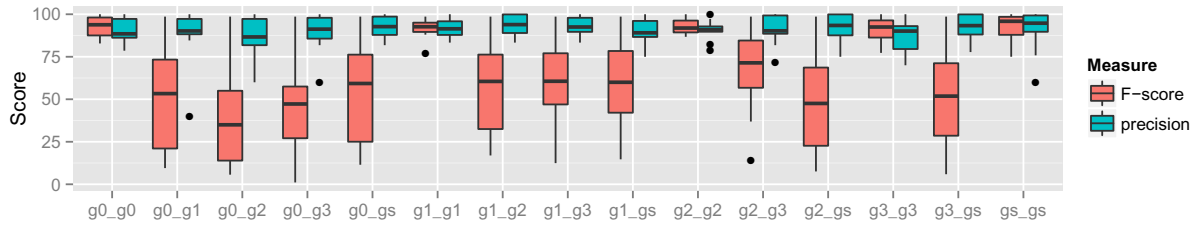

FIGURE 3.12. SLEUTH's code theft detection performance when the adversary's compiler is known, so the adversary is using optimization levels to hide his theft; here, we are computing *Sleuth*$(5, \langle 1, 60\%, 70\% \rangle, \{\texttt{gcc}\}, \{\texttt{O0}, \texttt{O1}, \texttt{O2}, \texttt{O3}, \texttt{Os}\}, P)$ in Equation 3.6 to generate the sample sets of comparison results that underlie the box plots.

discussion of outliers below. More details on this imprecision can be found in the outliers section below. Across all the optimization pairs in Figure 3.12, the mean precision is 91.58% with a standard deviation of 1.61%. SLEUTH's precision is therefore very high when the compiler is known, but the optimizations used, by either the victim or the suspect are not. The mean F-score is 60.04% with a standard deviation of 19.77%. This demonstrates that, when the compiler is known, SLEUTH's recall is also reasonably high.

3.5.6.2. *Compiler Resilience.* To shed light on SLEUTH's performance against different compilers, while controlling for optimization, we ask "Can SLEUTH detect code theft when the thief uses different compilers to

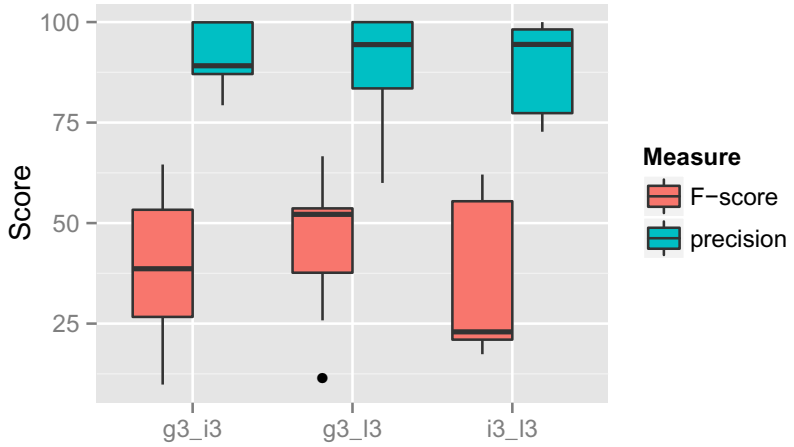

FIGURE 3.13. SLEUTH's detection performance when the adversary varies the compiler, while using the compiler's most aggressive optimization level, to hide his theft; here, we compute $Sleuth(5, \langle 1, 60\%, 70\% \rangle, \{\texttt{llvm}, \texttt{gcc}, \texttt{icc}\}, \{\texttt{O3}\}, P)$ in Equation 3.6.

hide the theft?". Rather than pick an optimization level uniformly, we choose `O3` as it is the most aggressive optimization, applying the most rewritings, and therefore intuitively will exaggerate the differences in the binaries the compilers produced. To answer this question, we considered the three most popular C/C++ compilers that run on Linux: `llvm`, the Intel C Compiler (icc), and `gcc` and compute Equation 3.6 with $Sleuth(5, \langle 1, 60\%, 70\% \rangle, C, \{\texttt{O3}\}, P)$.

Figure 3.13 shows the results for the 3 (3 choose 2 without replacement) combinations. In the figure, l is `llvm`, i is icc, and g is gcc. Each data point is the sample set of comparisons of all functions in V against all functions in S, where different compilers produced V and S. For instance, g3_i3 compares a program compiled with `gcc` and the O3 flag, with the same program compiled with icc and the O3 optimization flag. As in the previous experiment, each box plot captures the results of comparing each $v \in V$ to each $s \in S$.

In this experiment, SLEUTH's mean precision is 88.31% with a standard deviation of 0.95% and its mean F-score is 38.33% with a standard deviation of 1.92%. The precipitous drop in F-score is due to the recall dropping from a mean of 49.49% for different optimization levels with the same compiler, in the optimization experiment above, to 24.5% for `O3` with different compilers in this experiment. Any drop in recall in the semantic phase compounds because the syntactic function call measures rely on functional I/O equivalence to group similar functions under the same name. The box plots suggest that both measures drop. To confirm this observation, we hypothesized that the function similarity tests used to construct each graph are drawn from the same population. Then, we gathered 500 of these tests, uniformly at random

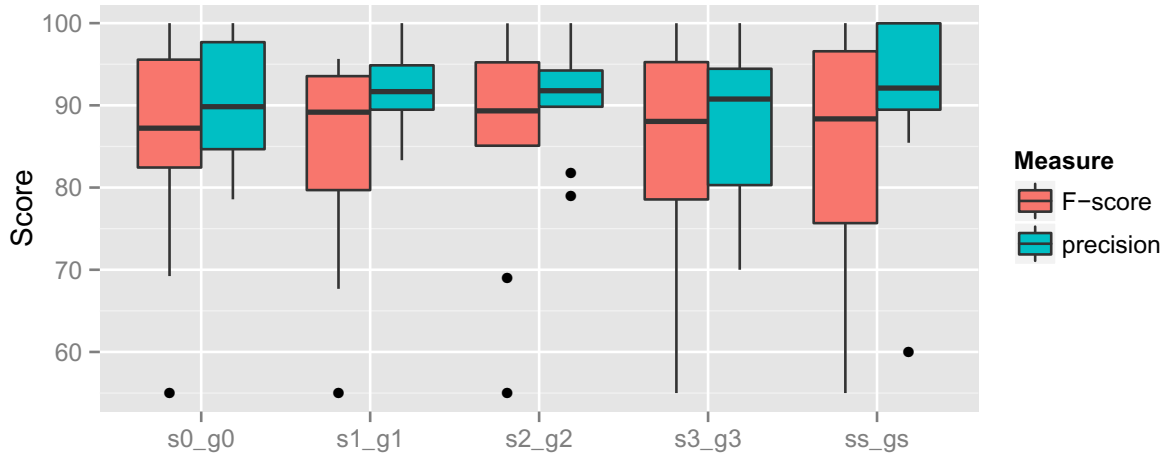

FIGURE 3.14. SLEUTH's detection performance when the adversary resorts to obfuscation, across all optimization levels, to hide his theft; here, for each P in our 10 benchmarks, we consider $V = \texttt{gcc}(x,P)$ and $S = \texttt{gcc}(x,\texttt{stunnix}(P))$ when computing $\mathit{Sleuth}(5,\langle 1,60\%,70\%\rangle,\{\texttt{gcc},\texttt{gcc+stunnix}\},\{x\},P)$ in Equation 3.6, $\forall x \in X$.

with replacement 100 times to produce a set of 100 precision and F-score results over the data from each figure. We applied Cliff's δ[1] to these sample sets: precision is indistinguishable across Figure 3.12 and Figure 3.13, with the p-value at 0.11 while the drop in F-score, however, is confirmed with a p-value less than 1e-4. Thus, SLEUTH has more difficulty with false negatives across compilers than optimization levels. Evidently, compilers produce binaries that are more different than those produced by a single compiler at different optimization levels.

3.5.6.3. *Obfuscator Resilience.* Next, we ask "Is SLEUTH resilient to source or binary obfuscators?". This experiment effectively asks how well SLEUTH does against a stronger adversary, one willing to augment compiler and optimization rewritings with obfuscations, transformations designed to obscure and thwart analysis. We extensively searched for obfuscators that support Linux. While we found many for Windows, we found only two binary obfuscators for Linux: binobf [63] and Loco [58]. Neither works on any recent Linux distribution. Binobf only works for programs compiled with `gcc` 3; it segfaults on more recent versions of `gcc` because it does not support Elf32_Rela relocations found in those versions. Loco no longer builds and has been abandoned. Therefore both are impractical for our adversary, who is stealing functions to save effort. Turning to source code obfuscators for Linux, we found only three. Of them, only the Stunnix [82] offers an evaluation version; Morpher and CodeMorph do not.

[1]Cliff's δ is an ordinal, robust, nonparameteric group difference test [27]. We used here because we cannot assume our data is normal as the studnet t-test requires and Mann-Whitney-Wilcoxon fails due to the fact that our scores saturate at 100 causing many ties.

We wished to study the effect of obfuscation, so, as in the first experiment, we fixed the compiler to `gcc`, the most popular Linux compiler. For all 5 (5 choose 2 with replacement) levels of optimization, we compiled each program in our corpus to produce an unobfuscated victim $V = \texttt{gcc}(x, P)$ and an obfuscated suspect $S = \texttt{gcc}(x, \texttt{stunnix}(P))$. We then compared each function in V against all the functions in S, as usual in these experiments. Thus, each data point is an unobfuscated, potential victim function compared against an obfuscated version of itself in the suspect, at the same optimization level.

Figure 3.14 shows the results. In the figure, g is `gcc`, and s is `gcc +stunnix`. For example, g0_s0 compares a program compiled with `gcc-O0` with and without the application of `stunnix`' obfuscations. The results are quite high despite obfuscation: the mean precision is 89.35% with a standard deviation of 2.28% and the mean F-score is 80.91% with a standard deviation of 0.76%. Further, the means of precision and F-score are quite close, so SLEUTH's recall is quite high. These results are quite similar to those in Figure 3.12, at the same optimization level. To confirm this hypothesis we compared the five medians in Figure 3.14 against the five medians of in Figure 3.12 whose optimization levels match. For both measures, we used Cliff's δ to determine whether the two sets of measures are distinguishable. For precision, the p-value 0.65 is evidence that `stunnix` has no more impact on SLEUTH's precision than varying optimization level. For F-score, the p-value is 0.032: the `stunnix` significantly reduced SLEUTH's recall and therefore its F-score. Interestingly, despite the fall in recall, SLEUTH's specifity remains high at 99.94% with standard deviation 0.01%. Thus, we can infer that obfuscation has little impact on SLEUTH's precision, but more impact on its recall, but not to a statistically significant degree. In short, these results show that `stunnix` is less effective against SLEUTH than changing compilers or varying optimization level. The outliers share the same root cause and, as before, we discuss them below.

To ease the automated identification of SLEUTH's performance, we turned off `stunnix`' symbol obfuscation, so we could continue to rely on function names to determine ground truth equivalence. However, SLEUTH does not depend on symbols to work correctly, since it uses functional I/O equivalence to compute its own names for functions that it uses when computing its syntactic measures. To validate this claim that symbol obfuscation does not affect SLEUTH, we uniformly at random picked 5 function pairs that were true positives in Figure 3.14. For these 5, we re-ran the experiment with `stunnix`' symbol obfuscation enabled and computed the confusion matrix: SLEUTH correctly equated each of the 5 pairs.

3.5.6.4. *General Resilience.* We have shown how well SLEUTH performs against optimizations, compilers, and obfuscations. We close with the question "How well does SLEUTH perform in general in

a free-for-all against any combination of its adversaries?". To answer this questions, we vary the compiler and optimization flags. To this this end, we compute Equation 3.6 with *Sleuth*(5, $\langle 1, 60\%, 70\% \rangle$, {`gcc`, `icc`, `llvm`, `gcc+stunnix`}, {`O0`, `O1`, `O2`, `O3`, `Os`}, P). For 30 of the 150 = (4 choose 2 = 10)(10 = 5 choose 2), both with replacement, pairs of compilers and optimizations, we compiled each program in our corpus to produce 10 pairs of V and S, as usual.

The resulting measurements are a mean precision of 88.12% with standard deviation 2.58%, and mean F-score is 36.21% with standard deviation 4.59%. This F-score result is closer to the compiler as an adversary than optimizations as an adversary because it is more likely that a different compiler created each function in a victim, suspect pair than that the same compiler created them. We tuned SLEUTH for precision, not recall. This experiment demonstrated that fact. In this, the hardest scenario when it knows least about the compilers used by either the victim or the suspect, its precision is just shy of 88% which is a strong defense against a victim's accusation being dismissed as frivolous, and, during a law suit, SLEUTH's precision will be useful for compelling discovery. Moreover, these results show how to improve SLEUTH's performance: reduce the search space by working to determine the compiler and optimization used by both the alleged victim and suspect, either via discover or compiler and optimization identification heuristics applied to the binary. Finally, its specificity remains high at 99.97% with standard deviation 0.03%.

3.5.6.5. *Outliers.* The outliers in all of these experiments are all comparisons of functions in our libgit, redis, yajl, sundown, and slash benchmarks. In all cases, the problem is false negatives. We have identified two reasons that account of all of the outliers we examined: imprecision in our current realization of lazy memory which currently does not distinguish between values and pointers and different inlining behavior at different optimization levels. The problem with lazy memory afflicts the functional equivalence heuristic, undermining the quality of the function names forwarded to syntactic measures.

An example function that shows the imprecision in the lazy memory is git_refspec__serialize from libgit2, compiled with the `Os` and `O3` optimization. Under Os, git_refspec__serialize writes through a pointer to a struct and then reads from a different struct. Under `O3` these structs are laid out as a union and partly overlap in memory, so, when it reads from the second struct, it finds the value from its previous read of the first stuct and does not consume a random value. This makes the function under `Os` look output-dissimilar from itself under `O3`.

The function ngx_chain_writer from nginx illustrates the problem with inlining. Under `O3`, the compiler inlines the functions ngx_debug_point, abort and raise, but not under `Os`. Another example is rewriteConfigFormatMemory

from redis, compiled with `O0` and `O3`. Under `O3`, __divdi3 and __moddi3 functions are inlined. Because of the inlining, the static function call measure falsely differentiates rewriteConfigFormatMemory from itself.

3.5.7. Limitations. We tested SLEUTH against the 10 of the most popular C programs on GitHub in Fall 2013 that effortlessly compiled (Section 3.5.2). Nonetheless, we face the standard external threat that our results may not generalize beyond our test set. When implementing SLEUTH, however, we carefully separated our training (Section 3.4.5) and testing set (Section 3.5.1).

Like any product of a large engineering effort, SLEUTH can be improved. It does not analyze multi-threaded programs. Instead, it approximates the semantics of a multi-threaded program; it converts all system calls, including threading functionality like fork spawn and exec, into NOPs. Nondeterminism in memory reads disrupts the equivalence of the inputs feed to two functions; nondeterminism in scheduling confounds the dynamic path measure. SLEUTH cannot distinguish pointers from values, and generates values in a bounded range (Section 3.4.3). If the code assumes a structure of a certain size, *e.g.* a `struct`, then assigning pointers values too close to each other might cause two `structs` to overlap and "share" memory values. Random values are consumed the first time a memory location is read, so different numbers of random values are consumed when `struct` do or not not overlap. This is the most common cause of SLEUTH's imprecision.

Function inlining, which varies with optimization levels, is another source of imprecision. It is especially acute when comparing conservative optimization, `O0`, against aggressive, `O3`, or when a function calls five or fewer functions. SLEUTH's current handling of unlinked external function calls as random number generators (see Section 3.4.3) can generate garbage behavior and cause both false positives and false negatives. This is especially problematic for functions where most if not all of its I/O behavior occurs in an external function. Functions that exhibit low entropy in their I/O behavior, *e.g.* return true or false, over most of their input domain are another source of false positives.

Our adversary steals functions for their behavior, so he preserves their semantics. In this evaluation, we have realized our adversary as an optimizing compiler and as a source-to-source obfuscator. We have not considered manual obfuscation and refactorings because they undermine the value of the theft, demotivating our adversary who seeks to save effort. Even with tool support, refactoring can be more expensive than manual rewriting [85]. Clearly, SLEUTH will benefit from more testing against a wider variety of adversaries, such as binary obfusactors or obfuscators that introduce a function dispatcher scramble calls. Nonetheless, we contend that our approach is robust against any semantics-preserving adversary. Consider our dynamic function call syntactic measure: intuitively, it would seem that even an optimizing compiler would disrupt it, but, in our test set, application functions are effectively immutable, other than inlining. We see this in

Figure 3.6, where the highest precision and F-score are achieved when the least editing, and therefore function call reordering, is needed to equate the victim and the suspect's output. Moreover, SLEUTH's inbuilt reliance on edit distance enables it to resist function splitting or fusing. For instance, if the original is *abc* and this becomes ab_1b_2c or $a(bc)$, where (bc) is the fusion of b and c, then these examples are two edits from the original.

3.6. Related Work

Two distinctive lines of research into clone detection exist — syntactic and semantic. However, we are the first to create a hybrid semantic and syntactic detector that pipelines semantic and syntactic measures, with the semantic restricting the syntactic to functionally similar code over which the syntactic acts to remove false positives.

Syntactic Syntactic code theft detection techniques fall into two categories: instruction signatures and control-flow.

In previous work [69], we created instruction signatures by extracting syntactic features to detect binary clones. This algorithm slides a window over a binary, normalizing the registers and memory locations within window, then clusters the normalized binary snippets with LSH to discover clones. We did not design this algorithm to handle compiler optimizations, let alone obfuscations, but it remains the state of the art of binary clone detection. As shown in Section 3.5.4 this approach is susceptible to the compiler as an obfuscator since it is detecting syntactic features. Schulman [73] applies related algorithms to our binary clone detection tool to find duplicated instruction sequences in a large set of binaries. Opcodes and API calls are used to find binary clones. Myles and Collberg [60] statically create birthmarks from k-grams of opcodes. Opcodes are only one feature of an instruction stream; the fact that compilers use of them is highly skewed exacerbates this information loss. Indeed, we invented normalized feature vectors in Sæbjørnsen *et al.* [69] precisely to find a more feature rich summary of an instruction sequence.

Hemel *et al.* [39] uses a more light-weight signature to detect code theft of files, by applying data compression. This works well for a few (< 100) large files, but as we show in Section 3.5.4 these heuristics are very imprecise when applied to fine-grained code theft.

Researchers have extensively studied control-flow signatures. Wang *et al.* [87] were, to the best of our knowledge, the first to use system call signatures to detect code theft, but Section 3.5.4 showed that 76.7% of all or 60% of functions with more than 100 instructions do not call any system calls so this approach cannot detect most function theft. Dullien *et al.* [30] and Flake [32] derive graphs from the syntactic structure of

binaries and compute graph isomorphism over a small set of malware programs. They determine similarity based upon function names and string references, which are easy to circumvent. SLEUTH uses semantic signatures to determine function similarity. Since they only evaluate on two pieces of malware and a patched DLL, it is not clear how well their approach would work in our context. Khoo *et al.* [49] detect binary clones using control-flow structure, data constant, or syntactic features. They introduce a syntactic feature extraction measure similar to that used in Sæbjørnsen *et al.* [69] and combine it with *k*-graphs extracted from the program's CFG. They report that these heuristics are effective against their adversary, the gcc compiler using only its `O1` or `O2` optimization levels.

Our approach complements control-flow signatures since we apply a function call measure as part of our hybrid semantic and syntactic measure. Unlike related work, we create a function call signature over all functions, not just system calls. This is important in our context since most functions does not call any system calls (see Section 3.5.4).

Semantic Previous research into semantic signatures rests upon either a full program trace or symbolic execution.

Jhi *et al.* [45] proposed to use core values as a birthmark to detect software plagiarism of executable programs. Their approach depends on taint-tracking inputs. To adapt their approach to function theft, they would therefore have to solve the hard problem of reaching each function. SLEUTH, thanks to its ability to construct an execution environment on demand, can begin execution from an arbitrary function's entry point. The authors report that their heuristic suffered from false positives when comparing short to long value sequences. In our corpus, this is a frequent occurrence: 66% of all function outputs had fewer than 5 values, while 21% had more than 10 values. Thus, their approach is likely to suffer from poor precision in our setting.

Gao *et al.* [34] use symbolic execution to create semantic signatures. Symbolic execution has well-known path explosion problems. In contrast, our semantic heuristic uses random testing and scales well. Gao *et al.*'s analysis efficiency drops when the number of semantic differences between binary files increases, because the graph isomorphism algorithm they use work best when two graphs are similar. When this assumption is not met, the authors recommend using a different graph isomorphism technique. In our context, the difference between the functions are most often very large since we compare arbitrary function pairs.

Chaki *et al.* [21] mix and match symbolic execution and syntactic heuristics. Against their adversary, Visual Studio with no optimization, they report that their heuristics do well over their benchmarks. Our adversary, in contrast, is one of several optimizing compilers and can employ source-level obfuscation.

Malware Detection Our work is also related to the large body of work on malware detection and analysis [15, 24, 25, 26, 40, 53, 92]. Here, the goal is to determine whether an unknown binary is malicious or not; and if malicious, whether it is a polymorphic or metamorphic variant of any of the known samples. *Signature-based* approaches use regular expression matching of binary code while *behavior-based* use runtime behavior, like system call sequences, for matching.

State of the art malware detection depends on a "zoo", large number of known malware samples, against which suspect binaries are compared. This zoo differentiates the malware identification and classification problems from code theft mining problem, where any function is a potential victim and any other function is a potential suspect. Further, the best signature-based approach described above uses a feature vector modeled on our previous work in Sæbjørnsen *et al.* [69] and, in Section 3.5.4, we showed this family of approaches is inefficient when detecting code theft hidden by an optimizing compiler. The state-of-the-art behavior-based measures will do no better, because they are exploiting the fact that a malware's payload must execute system calls to harm its victim. As we have shown in Section 3.5.4, most functions do not execute system calls so system-call signatures are insufficient for the code theft mining problem.

Function Identification Researchers have developed techniques, such as alias analysis [14] and interface identification [17], that are crucial for identifying functions in binaries. This problem is orthogonal to our goal of developing practical techniques for detecting binary similarity.

3.7. Conclusion

In this chapter, we have defined the binary code theft mining problem, and shown that neither semantic approaches, which have too many false positives, nor syntactic approaches, which have too many false negatives, can solve it alone. We have presented a novel hybrid semantic and syntactic formulation that relies on one approach to combat the weakness of the other. Our approach is robust in the fact of any adversary that preserves the semantics of the code it steals.

We realized SLEUTH and evaluated it on 10 popular C programs on GitHub in the fall of 2013. SLEUTH advances the state-of-the-art; no other theft detection tool scales to thousands of functions or has higher F-score and precision. SLEUTH beats compiler optimization, different compilers and source code obfuscators with high F-score and precision. In all our experiments, SLEUTH's specificity stays above 99.94%, significantly

in excess of the 94–97% specificity of mammograms [41] which is recommended every 2 years for women over 50. SLEUTH has a mean precision of 88.12% when it does not know which compilers and optimization levels are both unknown and a mean F-score of 91.03% when both are known. SLEUTH may transform IP litigation: pretrial, plaintiffs could use SLEUTH's high precision to defend themselves against dismissal of their complaint as frivolous and, during a lawsuit, to compel discovery.

CHAPTER 4

A Mixed Execution Framework for Binaries

We present SCOUT, an extensible abstract interpretation framework for binary analysis. SCOUT incorporates a loader to handle both dynamically and statically linked binaries and employs heuristics to analyze stripped binaries. The challenge of implementing binary abstract interpretation is interpreting instructions as operations upon a customizable abstract storage where pointers and values can be unknown. The core idea of this chapter, *mixed interpretation*, allows changing modes at each instruction to interpret the instruction concretely, symbolically or abstractly. SCOUT implements mixed interpretation and has the unique capability of executing a program concretely, symbolically or abstractly, *starting from any offset*. Thus, SCOUT can analyze program fragments or functions in third-party binaries.

The challenge inherent to starting from any offset is handling the initial lack of state. Current binary analysis tools do not support exercising a code fragment at an arbitrary offset with a randomly generated input. Instead they either statically analyze a decompiled form, execute a trace of the program concretely, or construct a symbolic state from a concrete trace. However, the requirement of analyzing a trace of a program precludes exercising a code fragment on a random input and because of that it is extremely challenging to separate the side-effects of a code fragment from the rest of the program. It also forces the user to think in terms of the whole program. To overcome the challenge of starting from any offset, SCOUT can lazily instantiate memory to values restricted by the information gleaned from the execution of the analyzed program up to the access of a memory location. The semantic clone detection implementation in Chapter 3 is a specialization of SCOUT that demonstrates some of its capabilities. SCOUT enabled us to create a semantic signature through exercising a code fragment with a randomly generated input and extracting the side-effects as output.

To test the correctness of the instruction implementation in SCOUT, we compared its concrete interpretation against native execution, instruction by instruction, for programs in POSIX.1-2008. For performance, SCOUT marshals a program's abstract state to and from native, concrete state when making systems calls; this mechanism passes the Linux Test Project [57]. SCOUT is easy to use: To incorporate a new abstract domain, one need only define a few functions and subclass two classes. SCOUT supports switching mode between multiple abstract domains.

4.1. Introduction

The core idea of this chapter is mixed interpretation. Mixed interpretation allows changing the interpretation mode on each instruction to symbolic, abstact or concrete execution. Mixed interpretation has two principal benefits: 1) it dynamically handles the state space explosion problem of symbolic execution by over-approximating a symbolic expression in an abstract domain, and 2) it enables a new framework to explore the semantics of binary fragments that we call SCOUT. A binary fragment is a disassembled fragment of a binary. It is common practice to explore the semantics of a fragment through studying its input-output behavior by running it concretely or symbolically. Exploring the semantics by running it concretely is difficult because it is hard to generate inputs that reach it; running it symbolically is also unlikely to reach it because of the state space explosion problem. Concretely starting execution at the start of the fragment quickly produces errors after a few instructions. One can symbolically execute a fragment, *i.e.* from an arbitrary offset. The challenge is to do this in a meaningful way. Mixed interpretation enables the developer to designate free variables as concrete, symbolic or abstract and bind values to them before interpreting the fragment.

Binary analysis frameworks are essential. Every program depends on external binaries, like the source language's standard library and runtime system. Many third-party applications and libraries are distributed without their source code. Binary analysis is needed to thoroughly analyze such programs. Even when source code is available, it is still desirable to analyze the executable code that actually runs on a system due to compiler errors and optimizations [7]. For these reasons, there has been burgeoning interest in binary analysis tools and frameworks, including McVeto [84], BINCOA [9], Jakstab [50], S^2E [22] and bitblaze [78].

When trying to read code and explore its semantics, it is common practice to experiment with the code by running it on various inputs to see what happens. Often, one wants to explore the semantics of a function or even a code fragment in isolation without having to solve the hard problem of driving execution to the start of the code in question. One may be interested in only some aspect of that code's execution, in other words, an abstract interpretation. Tool support for this practice has been neglected in general, and no work has addressed supporting it for binaries. Manually achieving these goals in binaries is tedious. Often, it is prohibitive to do the work to reach an arbitrary program point or construct the needed state to execute from that point. This work is the first to provide such framework support for binaries; our goal is to help automate exploring the semantics of binaries. We believe this capability will enable automating understanding binaries.

We present SCOUT, a binary analysis framework that, starting from any program point, supports the *mixed interpretation* of a binary, the capability of changing mode to interpret each instruction concretely, symbolically or abstractly. SCOUT is also the only framework that supports all of these modes. When

exploring code semantics, each of these interpretation modes has different strengths — each gives the user a different view of the code's behavior and thereby allow the user to crosscheck a hypothesis she may have formed about what the studied code is doing. The key difficulty of mixed interpretation is mixing under and over approximations. Mixed interpretation is semi-supervised and relies on its human operator to resolve the under/over conflict. SCOUT relies on a user-defined specification to dictate when the mode should be switched.

Concrete execution shines when a concrete memory state or trace is available. Loading, linking, and initializing (*i.e.* resources allocation such as libc initialization) a binary are canonical examples. Symbolic execution binds a symbolic expression to each symbolic variable and maintains a path constraint that defines the subset of the input domain that would drive execution to reach the current program point. Symbolic execution is not a panacea, however. It suffers from the path explosion problem, *i.e.* the fact that symbolic states grow exponentially with the number of conditional branches, and theorem prover limitations, *i.e.* the fact that satisfiability modulo theorem (SMT) solvers cannot solve some interesting constraints within a given resource bound. When symbolic execution is infeasible, concrete execution can be used to help make progress at the cost of imprecision. SCOUT, like its predecessors, BitBlaze [78], S^2E [22] and SAGE [36], supports both concrete and symbolic execution. However, it differs from these frameworks in its capability to interpret a code fragment at an arbitrary offset.

Abstract interpretation is a powerful framework for static program analysis [28]. An abstract interpreter evaluates a program in an abstract domain. A well-chosen abstract domain is critical to obtain a precise and tractable analysis. Designing a good abstract domain is important and orthogonal to our work. A common tactic is to capture partial knowledge of how a program works in an abstract domain. To support this tactic, SCOUT can abstractly interpret a program as much or as little as desired. Furthermore, SCOUT can abstractly interpret a program in multiple domains at the same time.

SCOUT combines a loader and disassembler with a per-program virtual machine (VM), or simulator. Unlike a system VM, SCOUT does not provide isolation, which means it does not have to contend with the *semantic gap*. Memory-isolated VMs, like VMWare, distinguish between host and guest VMs, which run in separate processes. The host VM interacts with the hypervisor to create and destroy guest VMs. Because host exists in a separate process, even when it can read the raw memory of a guest VM, it does not know how the guest VM is using that memory. This is the semantic gap.

Source-level analyzes are often stymied by calls to external libraries. Since it is a VM, SCOUT simply executes these libraries and therefore does not need labor-intensive models [8] or function summaries. This

strategy does not scale to system calls, where it would require the expensive simulation of the entire operating system. To handle a system call, SCOUT therefore, like S^2E [22], concretizes and marshals any abstract state needed to compute the calls parameters, makes the call natively, then unmarshals the result back into its abstract state. By default, SCOUT exits when it lacks sufficient state to concretizes a system calls parameters, but allows users override this behavior.

Machines implement instructions that operate on storage locations. Extensible VMs allow redefinition of instructions; SCOUT is the first to simultaneously support extensible abstract storage, where storage encompasses heap, stack and registers. SCOUT's abstract storage makes it convenient to implement abstract interpretation-based binary analyses: like other abstract interpreters, the programmer must redefine instructions to operate upon an abstract domain. Unlike others, SCOUT centralizes the redefinition of storage on which instructions operate. In other machines, abstract state must be handled on a case-by-case basis, out of band, in shadow storage maintained by the developer. It is SCOUT's realization of abstract storage that underlies its support for mixed interpretation.

Interpretation from Arbitrary Offset The key challenge when starting interpretation at an arbitrary instruction is the lack of initial state. Concretely interpreting a program fragment using random values segfaults very quickly because generating a valid stack, heap and register file is highly improbable. To solve this problem, SCOUT computes an abstract store from an arbitrary binary fragment from which it lazily instantiates concrete values that are consistent with the results of the interpretation up to the point of a storage access. It is this domain that allows SCOUT to begin interpretation at any offset. Repeatedly using SCOUT to interpret from a particular offset produces input-output pairs from which a programmer can infer that fragment's semantics. The fact that SCOUT can analyze binaries from any offset allows it to complement framework that operate on an intermediate representation constructed from source-code like KLEE [18], which, instead of abandoning an execution path upon encountering an external function, could hand off the analysis to SCOUT.

SCOUT's extensible support for abstract storage, which underlies its supports for mixed interpretation, is just one part of its general extensibility (Section 4.4.4). SCOUT allows its user to change the semantics of an instruction, or a sequence of instructions, via adapters and callbacks. Its user can also change semantics of system calls, and change semantics of reading and writing memory. All these contribute to SCOUT's ease of use for implementing binary analyses and running binary code fragments to learn about their behavior.

Contributions The work in this chapter makes the following three principal contributions:

```
1   void foo(int a, int b){
2     int a := c_i
3     unsigned int b := c_j
4
5     int c := a
6     while ( b != 0 )
7       if odd( b )
8         a += c
9
10      c <<= 1
11      b >>= 1
12
13    for i in 0..a
14     b++
15
16    c := a
17    for i in 1..100
18      if i mod 2
19        a := a + b
20      else
21        a := a − b
22
23    assert( b == 1000 )
24    assert( a != c )
25  }
```

FIGURE 4.1. Example that shows utility of Mixed Interpretation

(1) The design and implementation of SCOUT, an extensible binary analysis framework that features an extensible abstract storage and supports mixed (*i.e.* concrete, symbolic, and abstract) analysis of binary fragments;

(2) A lazily instantiated memory model that allows mixed interpretation from an arbitrary offset and therefore realized framework support for the common practice of running code to understand it; and

(3) An empirical evaluation that shows the engineering quality of SCOUT in terms of the correctness of its components (such as instruction simulation and system call marshaling), its performance, and its capacity to analyze a binary from any offset.

4.2. Illustrative Example

Figure 4.1 shows the importance of *mixed* interpretation. Under concrete execution alone, the true positive at the first assert is very unlikely to be triggered, assuming random inputs. Under abstract interpretation using the interval domain alone, both assertions, including the false positive, will trigger. The assertions will not be reached under symbolic execution alone due to the loop at 13–14. All three execution modes are needed to reach and trigger the assertions with high precision. Neither abstract interpretation nor symbolic execution

can handle the first loop: concrete execution alone can punch through it. We need the interval domain for the second loop; certainly symbolic execution explodes here. Finally, we need the precision of symbolic execution for the third loop.

The first loop computes the multiplication of the variable 'c' and 'b', and adds the value to 'a'. The multiplication is computed in the concrete interpretation because of the conditional on 'odd(b)'. Symbolically the conditional clause causes a state explosion problem. The interval domain does not help terminate this loop because the even conditional collapses the interval domain into single concrete values.

The second loop computes the variable 'a' that is used in assertion 1 and 2. Symbolically the unknown loop upper bound causes state explosion, and random testing is computationally expensive since it has to do 'a' iterations for every test. However, this loop terminates in a single iteration in the interval domain.

The third loop is evaluated symbolically. Evaluating this loop in the interval domain incorrectly produces a false positive in the first conditional because the lower bound of the interval 'a' is negative. In both the concrete and symbolic domain 'b' has no effect on 'a', a negative 'b' value is therefore not possible, but in the interval domain 'b' can be less than 0. However, the loop cannot evaluate in the concrete domain because doing so would probably not find 'b==1000' in the first assertion and would have a high likelihood of producing a false negative.

4.3. Formalism

Values in mixed interpretation (MI) can be concrete c, symbolic s, or abstract, where $a_i \in \mathscr{A}$ denotes a particular abstract domain in the set of supported domains $\mathscr{A}$. The set of domain tags is $\mathscr{D} = \{c, s, a_i\}$, These domains are disjoint; a particular variable or location can be bound to a value from only one of these domains.

$$D = D^d = D^c \uplus D^s \biguplus_{a_i \in \mathscr{A}} D^{a_i} \tag{4.1}$$

Conceptually, memory is an array of values. An address is an index into the array. To handle symbolic or abstract addresses, MI uses McCarthy expressions, which represent the set of concrete memories that satisfy a set of constraints on addresses and values [59]. Formally, a McCarthy expression is an equational theory that defines the set of possible concrete memories that a sequence of read, called **select**, and write, called **store**, operations can create. We write $d \in D$ to denote any value, when the domain is unimportant. When **select** : $m \times e \to D$ and **store** : $m \times e \times e \to m$, where Figure 4.2 below defines the syntax of the expressions

$$
\begin{array}{rcl}
e & ::= & x^d, \text{ for } x^d \in D^d \\
& | & v, \text{ for } v \in L \\
& | & e_1 \diamond^d e_2 \\
& | & \nabla^d e \\
& | & \textbf{select}(m, e) \\
S & ::= & S; S \\
& | & \textbf{store}(m, e_1, e_2) \\
& | & v := e, \text{ for } v \in L \\
& | & \textbf{goto } e \\
& | & \textbf{if } e_1 \textbf{ then goto } e_2 \textbf{ else goto } e_3 \\
& | & \textbf{assert } e \\
m & ::= & \textbf{store}(m, e_1, e_2) \\
& | & \varepsilon \\
\diamond^d & ::= & \text{binary operator} \\
\nabla^d & ::= & \text{unary operator}
\end{array}
$$

FIGURE 4.2. The syntax of MILANG, a minimal, assembly-like language used to illustrate MI; throughout the figure, $m \in \mathscr{D}$ is the domain annotation and L maps variables to values.

m and e and the expression m also obeys the equational axiom:

$$
\textbf{select}(\textbf{store}(m, e_1, x), e_2) = \begin{cases} x & \text{if } e_1 = e_2 \\ \textbf{select}(m, e_2) & \text{otherwise.} \end{cases} \tag{4.2}
$$

The fact that MI values carry domain annotations is orthogonal to their storage in memory encoded as a McCarthy expression.

MILANG. Figure 4.2 defines the syntax of a minimal language for illustrating MI, called MILANG. Here, we assume that an oracle supplies the state and operator domain annotations. $\diamond^d$ and ∇^d include the usual operators; ∇^d additionally includes an identity function, which, conjoined with its domain annotation, converts its operand from one domain to another. Figure 4.5 uses these metasyntactic functions and variables to define the semantics of MILANG below.

Figure 4.3 defines meta-syntactic functions and variables that MI requires. Abstractly, von Neumann machines provide three types of storage: the register file, the stack, and the heap. Registers are statically named in binaries, so MI represents them directly in L, the domain of Δ, the variable to value map that Figure 4.3 defines. In contrast, the stack and heap locations may be computed and may therefore be symbolic or abstract. To minimize MI's assumptions, MILANG does not distinguish between heap and stack, but

$$\Delta : L \to D \triangleq \text{maps a variable to its value}$$
$$\Psi : D \times \mathscr{D} \to D^n \triangleq \text{converts between domains}$$
$$\mathit{next} : D \to \mathbb{N} \triangleq \text{computes the next instruction}$$
$$M \triangleq \text{the McCarthy expression for both stack and heap}$$
$$\text{PC} \triangleq \text{the program counter, points to next statement}$$
$$\Pi \triangleq \text{the path constraint/condition}$$

FIGURE 4.3. Meta-syntactic functions and variables.

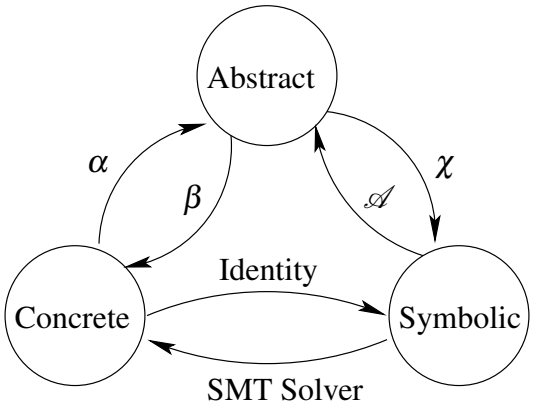

FIGURE 4.4. Domain conversions under mixed interpretation.

implicitly captures both into a single McCarthy expression, M in Figure 4.3. The PC is the program counter; it identifies the next statement to execute. The **goto** statement may bind a symbolic or abstract domain to the PC. When this happens, the *next* function extracts a concrete identifier for the next statement. Π is the standard path constraint/condition from symbolic execution. It is the conjunction of the control expressions, in MILANG if conditionals, from program start to the current program point.

MI evaluates operations in a single domain. Under MI, the execution *mode* of the last operation that produced the value written to a variable or location determines the domain of a value. When the domain of an operand to an operator mismatches that operator's domain annotation, MI converts it to the operator's domain prior to execution. Figure 4.4 shows the various transformations MI must support.

Transforming concrete values into an abstract domain is well-studied; indeed, their definition in terms of an abstraction function α and a concretization function γ underlies abstract interpretation. Similarly, transforming state between the concrete and symbolic state spaces is trivial (any concrete state forms a trivial symbolic expression) or well-studied in the form of SMT solvers.

We assert that abstract domains are less expressive than first order logic, and can thus be straightforwardly transformed into symbolic expressions; we denote this transform $\chi : D^{\mathscr{A}} \to D^{s}$. For MI, the author of an abstract domain must equip it with χ, just as she must define abstraction and concretization . In the sign domain, $\chi^{\text{sign}}(x = +) = x > 0$.

Algorithm 8 maps symbolic expressions to abstract domains: $\alpha_s : D^s \times \mathscr{A} \times D^{\mathscr{A}} \to D^{\mathscr{A}}$; $\alpha_s(\phi, a_i, a)$ returns $\bot_{a_i}$ if any call to the underlying solver, via solve, returns unknown.

input ϕ, an FO formula representing a symbolic state.
input a_i, an abstract domain that defines χ, α, and $\sqcup$.
input a, an accumulator in the target abstract domain.

if solve$(\phi \Rightarrow \chi(a))$ **then**
  return a
else /* Consider a fresh solution to ϕ */
  return $\alpha_s(\phi, a_i, \alpha(\text{solve}(\phi \wedge \neg\chi(a))) \sqcup a)$
end if

Algorithm 8 defines α_s whose goal is to compute $a = \alpha([\![\phi]\!])$; in theory, α_s always achieves its goal, since the solver would never run out of resources and return UNKNOWN. Intuitively, $\alpha_s(\phi, a_i, a)$ probes the solution space of the input symbolic expression ϕ, repeatedly applying α_{a_i}, the abstraction function of the domain a_i, to the growing subset of ϕ abstracted by $\chi_{a_i}(a)$, the symbolic expression of the abstraction a that α_s has constructed to contain the set of solutions to ϕ it has considered. A solution is found once ϕ implies $\chi_{a_i}(a)$, *i.e.* $\alpha_{a_i}([\![\phi]\!]) \sqsubseteq_{a_i} a$, as desired. To our knowledge, the mapping of a symbolic expression into a target abstract domain has not been studied.

Finally, we define ψ, the domain conversion function. Its definition uses choose : $2^X \to X$, a function that nondeterministically returns an element from the set X. It returns $\bot$, if its operand is $\emptyset$ or UNKNOWN, which an SMT solver returns when it exhausts its resources. The superscript f in y^f signifies that y is a unique, fresh variable, which therefore cannot collide with any other symbolic variable.

$$
\begin{array}{rrcl}
a_i \to c & \mathrm{add}^c(\psi([1,2]^{\mathrm{IV}},c),\psi([3,4]^{\mathrm{IV}},c)) & \Rightarrow & \mathrm{add}^c(1^c,3^c) \Rightarrow 4^c \\
s \to c & \mathrm{add}^c(\psi((x \le 1 \wedge x \le 2)^s,c),\psi((y \ge 3 \wedge y \le 4)^s,c) & \Rightarrow & \mathrm{add}^c(2^c,4^c) \Rightarrow 6^c \\
c \to a_i & \mathrm{add}^{\mathrm{IV}}(\psi(2^c,a_i),\psi(2^c,a_i)) & \Rightarrow & \mathrm{add}^{\mathrm{IV}}([2,2]^{\mathrm{IV}},[2,2]^{\mathrm{IV}}) \Rightarrow [4,4]^{\mathrm{IV}} \\
s \to a_i & \mathrm{add}^{\mathrm{IV}}(\psi((x \ge 1 \wedge x \le 2)^s,\psi(y \le 3 \wedge y \le 4)^s) & \Rightarrow & \mathrm{add}^{\mathrm{IV}}([1,2]^{\mathrm{IV}},[3,4]^{\mathrm{IV}}) \Rightarrow [4,6]^{\mathrm{IV}} \\
c \to s & \mathrm{add}^s(\psi(2^c,s),\psi(2^c,s)) & \Rightarrow & \mathrm{add}^s((a=2)^s,(b=2)^s) \\
 & & \Rightarrow & (a=2 \wedge b=2 \wedge a+b)^s \\
a_i \to s & \mathrm{add}^s(\psi([1,2]^{\mathrm{IV}},s),\psi([3,4]^{\mathrm{IV}},s)) & \Rightarrow & \mathrm{add}^s((x \ge 1 \wedge x \le 2)^s,(y \ge 3 \wedge y \le 4)^s) \\
 & & \Rightarrow & (x \ge 1 \wedge x \le 2 \wedge y \ge 3 \wedge y \le 4 \wedge x+y)^s
\end{array}
$$

TABLE 4.1. Expression evaluation under MI, where $\mathrm{add}^d \in \diamond^d$ is a binary operator and $a_i = \mathrm{IV}$, the interval domain.

$$
\text{(4.3)} \qquad \text{For } m,n \in \mathscr{D}, \psi : D^d \times \mathscr{D} \to D^n, \psi(x^d,n) = \begin{cases}
x^d & \text{if } m = n \\
y^f = x^d & \text{if } m = c \wedge n = s \\
\alpha_n(x^d) & \text{if } m = c \wedge n \in \mathscr{A} \\
\textbf{choose}(\{y^c \mid y^c \in \mathrm{solve}(x^d)\}) & \text{if } m = s \wedge n = c \\
\alpha_s(x^d,n,\bot_n) & \text{if } m = s \wedge n \in \mathscr{A} \\
\gamma_m(x^d) & \text{if } m \in \mathscr{A} \wedge n = c \\
\chi_n(x^d) & \text{if } m \in \mathscr{A} \wedge n = s \\
\bot & \text{otherwise.}
\end{cases}
$$

In concrete mode, operators have their standard logic or arithmetic semantics. In symbolic mode, they produce new symbolic expressions. As usual, every author of an abstract domain must, in addition to α and γ, define abstract operations for their domain; for MI, she must also define χ. Table 4.1 shows how the add instruction evaluates various operands for various combinations of domain annotations.

Operational Semantics The operational semantics in Figure 4.5 operate upon a domain-annotated abstract syntax tree (AST). Under symbolic execution, a particular **if** statement can generate two new states, one for each IF rule, when bindings can be found that set the conjunction of the if's conditional and the path condition Π both to true and to false. Under MI, **goto** is analogous. When its target expression defines a set of targets, the *next* function nondeterministically selects some subset of them. Each selection generates a distinct state for the interpreter to explore.

Thus, these operational semantics rules are not syntax-directed. The **if** and **goto** statements can be matched multiple times by the same rule, although each matching is distinct. For **if**, two rules may be

$$\frac{}{\langle x^d | M, \Delta \rangle \Downarrow x^d} \quad \text{VALUE}$$

$$\frac{x^d = \Delta(v)}{\langle v | M, \Delta \rangle \Downarrow x^d} \quad \text{VARIABLE REFERENCE}$$

$$\frac{}{\langle \mathbf{select}(M, e) | M, \Delta \rangle \Downarrow x^d} \quad \text{SELECT}$$

$$\frac{\langle e | M, \Delta \rangle \Downarrow x^d}{\langle \nabla^t e | M, \Delta \rangle \Downarrow \nabla^t(\Psi(x^d, t))} \quad \text{UNARY OPERATORS}$$

$$\frac{\langle e_1 | M, \Delta \rangle \Downarrow x_1^d \qquad \langle e_2 | M, \Delta \rangle \Downarrow x_2^e}{\langle e_1 \diamond^t e_2 | \mu, \Delta \rangle \Downarrow \Psi(d_1^d, t) \diamond^t \Psi(d_2^n, t)} \quad \text{BINARY OPERATORS}$$

(A) Expressions

$$\frac{\langle S_1 | M, \Delta, \Pi, PC \rangle \Downarrow M_1, \Delta_1, \Pi_1, PC' \qquad \langle S_2 | M_1, \Delta_1, \Pi_1, PC' \rangle \Downarrow M_2, \Delta_2, \Pi_2, PC''}{\langle S_1; S_2 | M, \Delta, \Pi, PC \rangle \Downarrow M_2, \Delta_2, \Pi_2, PC''} \quad \text{SEQUENCE}$$

$$\frac{M' = \mathbf{store}(M, e_1, e_2)}{\langle \mathbf{store}(M, e_1, e_2) | M, \Delta, \Pi, PC \rangle \Downarrow M', \Delta, \Pi, PC + 1} \quad \text{STORE}$$

$$\frac{\langle e | M, \Delta \rangle \Downarrow x^d \qquad \Delta' = \Delta[v \leftarrow x^d]}{\langle v := e | M, \Delta, \Pi, PC \rangle \Downarrow M, \Delta', \Pi, PC + 1} \quad \text{ASSIGNMENT}$$

$$\frac{\langle e | M, \Delta \rangle \Downarrow x^d \qquad PC' = next(x^d)}{\langle \mathbf{goto}\ e | M, \Delta, \Pi, PC \rangle \Downarrow M, \Delta, \Pi, PC'} \quad \text{GOTO}$$

$$\frac{\langle e_1 | M, \Delta, \Pi, PC \rangle \Downarrow \mathbf{T} \qquad \langle \mathbf{goto}\ e_2 | M, \Delta, \Pi, PC \rangle \Downarrow M, \Delta, \Pi, PC' \qquad \Pi' = \Pi \wedge e_1}{\langle \mathbf{if}\ e_1\ \mathbf{then\ goto}\ e_2\ \mathbf{else\ goto}\ e_3 | M, \Delta, \Pi, PC \rangle \Downarrow M, \Delta, \Pi', PC'} \quad \text{IF-TRUE}$$

$$\frac{\langle e_1 | M, \Delta, \Pi, PC \rangle \Downarrow \mathbf{F} \qquad \langle \mathbf{goto}\ e_3 | M, \Delta, \Pi, PC \rangle \Downarrow M, \Delta, \Pi, PC' \qquad \Pi' = \Pi \wedge \neg e_1}{\langle \mathbf{if}\ e_1\ \mathbf{then\ goto}\ e_2\ \mathbf{else\ goto}\ e_3 | M, \Delta, \Pi, PC \rangle \Downarrow M, \Delta, \Pi', PC'} \quad \text{IF-FALSE}$$

$$\frac{\langle e | M, \Delta, \Pi, PC \rangle \Downarrow \mathbf{T}}{\langle \mathbf{assert}\ e | M, \Delta, \Pi, PC \rangle \Downarrow M, \Delta, \Pi, PC + 1} \quad \text{ASSERT-TRUE}$$

$$\frac{\langle e | M, \Delta, \Pi, PC \rangle \Downarrow \mathbf{F}}{\langle \mathbf{assert}\ e | M, \Delta, \Pi, PC \rangle \Downarrow \bot} \quad \text{ASSERT-FALSE}$$

(B) Statements

FIGURE 4.5. Operational Semantics.

generated if the current state can find both a binding under which the **if**'s conditional is true and one under

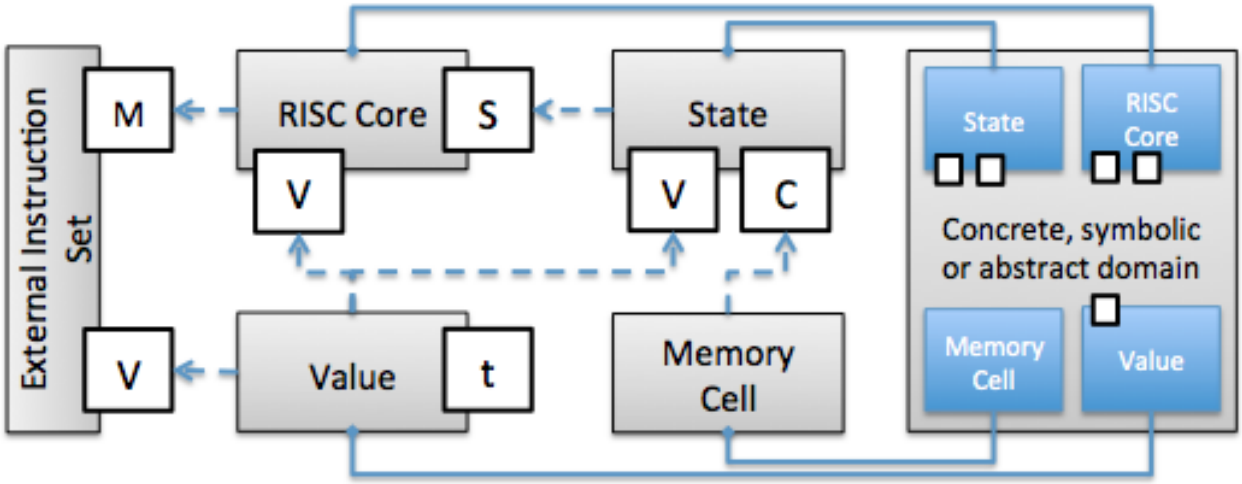

FIGURE 4.6. The architecture of SCOUT.

which it is false. For **goto**, as many rules as distinct binding for the PC that the next function generates. Each rule defines a state. These states grow with the number of paths, and suffer from the path explosion problem.

The PC in the sequence and assignment rules is concrete and thus can simply be incremented. The PC is always concrete; the next function ensures this. Conceptually, passing an abstract or symbolic value to the next function just simply spawns as many states as distinct PCs the next function extracts from its operand. When that set is large, or unbounded the implementor may define next to return $\perp$ and terminate execution or finitely sample from its operand.

4.4. Design and Realization

SCOUT is a simulator framework for any instruction set, but currently implements 32 or 64-bit, Intel x86 [44] and AMD [3] Linux programs, and runs on 32 or 64-bit Linux. SCOUT integrates the Yices SMT solver to reason about symbolic expressions [31]. Figure 4.6 depicts SCOUT's architecture. This section discusses each of the components in the figure. The two core components we discuss first are cross-cutting concerns not explicitly present in the figure: SCOUT's VM, which touches all the components, and SCOUT's abstract storage model, which unites the register file and memory. Two core principles guided our design: 1) ease of use and 2) mix and match customizability.

When using a VM to tackle binary analysis, one must decide how powerful and heavyweight to make the VM, how to implement abstract storage (Section 4.4.1) and whether and to what extent to allow manipulation of instructions (Section 4.4.2). To bootstrap interpretation, one needs a loader to read the specimen and its dependencies, like libraries. Binary executables are often dynamically linked. Unlike most disassemblers, SCOUT handles dynamic libraries. It read symbols from the binary that are not stripped and learns the binaries

layout, *i.e.* where the globals are. Linking recursively parses all shared object dependencies. Mapping chooses virtual addresses for parts of the binary file as if SCOUT were creating a new OS process. For instance, mapping an ELF file causes SCOUT to choose virtual addresses for all ELF segments. Relocation applies relocation fixups to patch pointers and offsets in various parts of the virtual address space.

SCOUT can control which memory the disassembler reads and disassembles. Unpackers are a classic example: SCOUT can simulate the unpacker and, when the unpacker is about to jump into the newly unpacked instructions, invoke the disassembler. The simulator can also provide the disassembler with a linked version of a program for disassembly. SCOUT is an extensible framework whose behavior can be modified in four ways: 1) instruction (re)definition, 2) system call (re)definition, 3) callbacks, and 4) adapters. Section 4.4.2 discusses instruction (re)definition; Section 4.4.4 discusses the last three mechanisms.

Terminology. The *specimen* is the program executing inside the simulator. We abbreviate operating system to OS and intermediate representation to IR. The *host OS* is the system for which SCOUT was compiled and on which the simulator is running. The *guest OS* is the environment (system calls, signals, *etc.*) provided by the simulator to the specimen. Following convention, we use VM to denote virtual machine. We use *functor* to mean a C++ functor, or function object, not category-theoretic functor. We use *storage* to refer to a machine's heap, stack and register file.

4.4.1. Abstract Storage. Storage systems and the instructions that operate on them must match: the instructions must read and write entities that the storage can contain; to turn it around, storage must contain entities on which instructions can operate. Achieving this match is the core design problem that abstract storage presents. To simplify this problem and to facilitate supporting many concrete ISAs, SCOUT realizes its abstract storage in terms of its IR.

Existing VMs implement instruction that operate on concrete program states. Unfortunately, concrete program states are sometimes unavailable. SCOUT implements abstract storage to handle this case. When a concrete state is unavailable, SCOUT can interpret instructions abstractly. This design decision means that SCOUT is the machine, which obviates recompilation. Abstract storage contains the effect of an instruction's operation upon storage. Concrete memory maps virtual addresses to bytes. Symbolic or abstract memory must be defined in conjunction with symbolic or abstract instruction semantics in order to store sensible values and handle cases, such as unknown addresses or values.

To create a user-defined domain, a programmer need only redefine at most a dozen functions, not all the instructions in SCOUT's IR. In SCOUT's abstract storage, values are expression trees.

```
1  template<size_t Len>
2  ValueType<Len> add(
3      const ValueType<Len> &a, const ValueType<Len> &b
4  ) const {
5      if (a.sign==ZERO && b.sign==ZERO)
6          return ValueType<Len>(ZERO);
7      if (0==(a.sign & NEGATIVE) && 0==(b.sign & NEGATIVE))
8          return ValueType<Len>(POSITIVE);
9      if (0==(a.sign & POSITIVE) && 0==(b.sign & POSITIVE))
10         return ValueType<Len>(NEGATIVE);
11     return ValueType<Len>();
12 }
```

LISTING 4.1. Redefinition of add for the sign domain.

A domain is represented by a Policy class. Because each policy has its own distinct instance of memory, multiple policies can be used concurrently. To achieve this isolation, SCOUT separates the specimen's address space from SCOUT's. We see this in action in the illustrating example in Section 4.2 where three different memories are used concurrently; the program is interpreted concretely until the entrypoint of main where we initiate a symbolic interpretation of a code fragement. A sign analysis in the sign domain rides above the symbolic interpretation.

State in Figure 4.6 represents the entire state of the machine (instruction pointer, registers, registers, control/status flags and core memory). The heap location in state is represented by MemoryCell. MemoryCell stores an address and data for a location. A ValueType is the values stored in registers and memory and used for memory addresses. In addition to holding the data for a memory location a ValueType defines basic operations upon memory ('==', '!=', and '<').

A single RISC machine can store and operate upon multiple memory states at the same time. In the standard configuration there are two states; the origin machine state and the current machine state.

4.4.2. Instructions. The core problem one confronts when defining extensible instruction semantics is the other side of the matching problem that abstract storage presents: any tool that interprets program semantics must accurately model how instructions operate upon the program state. Consonant with our core design principle that SCOUT be extensible, SCOUT allows a developer to change or manipulate instruction semantics by changing either 1) how an interpreted instruction is translated into our RISC-like instruction set or 2) how SCOUT interprets an instruction in its IR, such as redefining the domain upon which an instruction operates. Listing 4.1 shows the redefinition of the add instruction to support sign analysis.

Most VMs or emulators choose to translate between the interpreted instruction set (*e.g.* x86) and an IR in a RISC-like instruction set to simplify manipulating instruction semantics and reduce the chance of

```
1  case x86_lea: {
2      if (operands.size()!=2)
3          throw Exception("instruction must have two operands", insn);
4      write32(operands[0], readEffectiveAddress(operands[1]));
5      break;
6  }
```

LISTING 4.2. Translating lea to SCOUT's internal IR.

implementation bugs. SCOUT is no exception to this rule. Listing 4.2 shows the translation of lea into SCOUT's internal instruction set.

SCOUT currently supports 110 instructions[1], about 17% of the total in the x86 instruction set [90]; it treats unsupported instructions as no-ops, since this allows SCOUT to ignore instructions that provide inessential functionality. Broadly, SCOUT simulates almost all integer instructions in x86 up to the 386 instruction set. SCOUT also simulates instructions required by the dynamic linker and glibc, such as the MOVD, MOVQ and two MMX instructions.

4.4.3. System Call Marshaling. SCOUT's abstract storage necessitates special-handling of system calls. The problem is translation from a specimen's internal, arbitrarily abstract state running under SCOUT into the concrete parameters that a system call requires. To solve this problem, SCOUT marshals the state necessary to perform a system call to and from a specimen's abstract storage. Currently, SCOUT only performs a system call if enough state exists in a specimen's abstract storage to provide the native system call with its arguments and exits otherwise, since system calls are almost always essential. If enough state does not exist, the user must either treat the system call as a no-op or provide an alternate mechanism for constructing the necessary state.

SCOUT is designed to simulate a specimen such that its abstract storage is equivalent to the same abstraction applied to the concrete memory the specimen produces when running natively. The fidelity of SCOUT's marshaling enables the specimen to interact with the host OS and programs running natively through system calls. SCOUT handles the following categories of system calls: file system, inter-process communication, memory management, memory maps, process properties (*e.g.*, getuid, setpgid, setrlimit), signals, socket calls, standard I/O, threading and time.

Listing 4.3 shows how to redefine sys_chown (#182) to return the "not implemented" error. To completely remove a system call implementation and cause the simulator to dump the specimen's core if it tries to invoke that system call, one has only to remove all enter, body, and leave callbacks for that system call.

[1] Section 4.5 explains how these 110 instructions were chosen.

```
1  class NoOp: public RSIM_Simulator::SystemCall::Callback {
2  public:
3    bool operator()(bool b, const Args &args) {
4      args.thread->tracing(TRACE_SYSCALL)->more(
5        "[NOOP]"
6      );
7      args.thread->syscall_return(-ENOSYS);
8      return b;
9    }
10 } syscall_noop;
11
12 RSIM_Simulator *sim = ...;
13 sim->syscall_implementation(182)->body.clear().append(
14   &syscall_noop
15 );
```

LISTING 4.3. Redefining a system call

4.4.4. Customization. The challenge is to provide customization in a principled fashion. One must decide how much customization to allow (should a subset of the simulator's behavior be inviolate?), how do customizations compose/interact and whether to support dynamic customization. SCOUT is relentlessly customizable; it is thread-safe and supports both compositional and dynamic customization. SCOUT provides four mechanisms to realize customization — instruction (re)definition, system call (re)definition, and modifiers. Instruction definition is discussed above in Section 4.4.2. SCOUT's system call mechanism that allows a user to augment or replace system calls; system call customization is particularly useful for handling system calls that require state than abstract storage defines at the point of the call. Many of the events that occur during the simulation can be tied to user-defined modifiers invoked before and/or after the action. Modifiers augment, replace, or skip the normal processing depending on the situation. A program can attach and detach modifiers at any point to temporarily change behavior. For instance, a pre-instruction modifier can check for unsupported instructions and skip them, or a pre-system-call modifier can count the total number of system calls executed per thread.

Listing 4.4 shows the definition of a modifier that prints "[FIRST CALL]" the first time a particular system call executes, then removes itself on line 10. To bind this modifier to system call 3, the programmer writes `sim->syscall_implementation(3)->enter.append(&notifier)`

Listing 4.5 shows a callback that implements instruction granular traces. In SCOUT's AST, the function that contains an instruction is the instruction's grandparent; the callback leverages this fact. The developer attaches this callback to the simulator with `add_insn_callback(RSIM_Callbacks::BEFORE, new ShowFunction)` and a snippet of its output is

```
1  class NotifyOnce :
2    public RSIM_Simulator::SystemCall::Callback {
3  public:
4  bool operator()(bool b, const Args &args) {
5    args.thread->tracing(TRACE_SYSCALL)->more(
6      "[FIRST CALL]"
7    );
8    RSIM_Simulator *sim =
9      args.thread->get_process()->get_simulator();
10   sim->syscall_implementation(args.callno)->enter.erase(this);
11   return b;
12 }};
```

LISTING 4.4. Modifier that prints the first use of a system call.

```
1  class ShowFunction: public RSIM_Callbacks::InsnCallback {
2  public:
3    virtual bool operator()(bool prev, const Args &args) {
4      SgAsmBlock *basic_block =
5        isSgAsmBlock(args.insn->get_parent());
6      SgAsmFunctionDeclaration *func =
7        basic_block ?
8        SageInterface::getEnclosingNode<
9          SgAsmFunctionDeclaration
10       >(basic_block) : NULL;
11     if (func && func->get_name()!=name) {
12       name = func->get_name();
13       args.thread->tracing(TRACE_MISC)->mesg(
14         "in function \"%s\"", name.c_str()
15       );
16     }
17     return prev;
18   }
19 private:
20   std::string name;
21 };
```

LISTING 4.5. Instruction granular tracing.

```
28129:1 0x0805e7c0[256]:   in function "__uname"
28129:1 0x0805e7cd[260]:   uname[122](0xbfffddb6) = 0
...
28129:1 0x0804c375[348]:   in function "__libc_setup_tls "
28129:1 0x0805f5c8[376]:   in function "__sbrk"
28129:1 0x080871f0[387]:   in function "brk"
28129:1 0x080871fe[393]:   brk[45](0)  = 0x080d5000
```

	Total	Simulator
System calls	293	110
Instructions	663	115

TABLE 4.2. Number of system calls and instructions whose semantics under SCOUT's simulator were verified.

The "28129:1" means the main thread of process 28129. The hexadecimal number is the address of the executed instruction followed by the value of the instruction counter in square brackets. The "in function" output includes the system call along with its arguments and return value.

4.5. Evaluation

We evaluate SCOUT along two dimensions. First, we evaluate the correctness of its implementation of instruction semantics and its system call marshaling mechanism, then present SCOUT's performance when used merely as a concrete simulator. Second, we describe how to use SCOUT to jump to an arbitrary offset. The arbitrary offset scenario shows how, especially for a large program like vim, SCOUT's support for interpreting from an arbitrary offset mitigates its simulation overhead. The evaluations described herein ran on a 12-core Intel Xeon X5680 3.33GHz with 48GB memory running Linux 2.6.32-5-amd64.

4.5.1. Correctness and Performance. Table 4.3 shows how many of the total system calls and instructions we have implemented. System calls do not make sense for the disassembler as interrupts trigger them. SCOUT implements all integer instructions that operate on integer registers in the 386 architecture, except those instructions introduced in 386 that operate on control and debug registers, such as dr0 or dr1. POSIX.1–2008 [64] specifies 161 system utilities, and is part of the Single UNIX Specification (SUS). These utilities can be found on most UNIX-like operating systems. These utilities interact with and change system state using system calls, and are likely to exhibit more complex system call behavior than most programs. Because of this we decided to implement the necessary functionality to support these utilities one-by-one. We currently support 59 of these utilities. The 110 instructions SCOUT current supports are precisely those needed to run these 59 utilities. We believe the instruction usage patterns of these utilities to be representative and that SCOUT can already support wide variety of applications. Development of SCOUT is ongoing; support for additional instructions is added as needed.

Verifying* SCOUT*'s Interpretation of Instructions A simulator must correctly implement instruction semantics, which often have complex conditional behavior. The core idea is to run SCOUT in parallel with a native execution of the executable SCOUT is analyzing and compare the input-output results of each executed

	Total Verified	Errors Reported	Actual	CI
diff	454822	490	0	22.84
grep	470688	1728	0	23.35
gzip	235550	298	0	7.54
ls	454760	614	0	22.99
patch	475613	1048	0	23.22

TABLE 4.3. Verification of instruction semantics: Confidence level is set to 99% and except for gzip the sample size is 30. For gzip we analyzed every single reported error. 50% percentage was used to calculate the confidence interval.

instruction. The specimen is run under a very simple network-based debugger which, in turn, runs under SCOUT.

To ease this task, we implemented a semantic analyzer. This tool seeks to verify that SCOUT's interpretation of an instruction's semantics accurately reflects how the instruction executes on real hardware. The semantic analyzer queries the debugger for the value of every register and memory read operation and stores the result.

SCOUT parses the executable and obtains the addresses of all executable segments. It then sets breakpoints for all addresses in those segments and asks the debugger to continue execution. When a breakpoint is reached, SCOUT looks up the SgAsmInstruction at that address. SCOUT queries the debugger for the instruction, then submits it to the semantic analyzer. SCOUT asks the debugger to single step and then compares writes, from previous step, with values queried from the debugger. SCOUT and the debugger then advance to the next breakpoint and repeat the process.

This evaluation methodology is conservative. First, the verifier does not handle interrupts. Therefore, instructions like "INT" will execute in the natively-running version (under the debugger) but not in the SCOUT's simulator. Therefore, one should expect the state after interrupts to be different. Second, instructions that interact with the environment, like RDTSC (read timestamp counter) return different values in different executions, such as native and simulated runs.

Since the goal of evaluation is instruction coverage, not specimen path coverage, we use a minimal input. For instance, the gzip test is executed as:

```
echo > gzip.input
gzip < gzip.input >/dev/null
```

We began our evaluation by manually inspecting every discrepancy between the native and simulated output reported by the semantic analyzer for gzip. There were 498 such failures, 183 of which are due to

	Total Number	Number Supported	Number of Tests	Tests passing natively	Tests passing	Tests failing
System calls	293	110	339	337	331	8
pthread routines	104	104	417	408	408	9

TABLE 4.4. Statistics about instructions and system calls supported by SCOUT.

system calls. The verifier does not handle system calls, so state always mismatches and generates a failure after an "INT 0x80" (183 calls). Likewise, there are 17 failures due to SYSENTER. After eliminating these two sources of failure, 298, the number reported in Table 4.3, remain. In addition to the system call and SYSENTER errors themselves, the instruction that follows them also generates an error, resulting in 200 errors. For those cases (approximately 50) we inspected, the only difference in state is the "ORIG_EAX" pseudo-register not accounted for by our semantic analyzer. This is expected, and we can ignore this failure. Of the remaining 98 errors, 48 failures are in the simulation of the MOV instruction. The EIP register had a simulated value of 0x40000ff7 after the instruction, but the debugger reported 0x40000ff0. In this case, the debugger is wrong because the instruction was at 0x40000ff0, so the EIP point to an address after it. The final 50 failures occur because the verifier currently does not support discovering the base address for the GS register. Basically, a memory reference is something like GS:[offset], where GS is a segment register and "offset" is a memory address, like 0x08040123. Segment shadow registers are internal to the x86 CPU — they are not externally accessible.

After our thorough manual inspection of one program, we then manually inspected 30 reported failures, chosen uniformly at random, to verify the instructions of the remaining programs. Table 4.3 reports the resulting interval into which the actual error falls with 99% confidence.

System Calls The Linux Test Project (LTP) [57] has the goal to deliver test suites to the open source community that validate the reliability, robustness, and stability of Linux. The LTP test suite contains a collection of tools for testing the Linux kernel and related features. LTP tests both passing and failing system call behavior. SCOUT uses LTP to test its system call marshaling; thus, SCOUT uses, and passes, exactly the same test suite used to test the system call implementation of the Linux kernel itself. To ensure that SCOUT stays conformant, we use continuous integration to enforce that all versions of SCOUT pass these unit tests.

For threading, we used the pthread conformance tests in the POSIX Test Suite to perform conformance, functional, and stress testing of the IEEE 1003.1-2001 System Interfaces specification in a manner that is agnostic to any given implementation. We apply the unit tests of all 104 pthread routines.

Table 4.4 reports 9 pthread unit test failures. One of the tests, thread_create.10−1.c, exhibited nondeterministic behavior both natively and under SCOUT. Four pthread_mutexattr tests did not compile, generating undeclared

		completion time	System Calls total	unique	Instructions total	unique
gzip	native	4ms	203	17	5.7m	–
	Valgrind	0.4s	–	–	5.7m	–
	SCOUT	9.4s	203	15	5.7m	73
diff	native	3ms	260	21	2.3m	–
	Valgrind	0.5s	–	–	2.3m	–
	SCOUT	4.1s	264	21	2.3m	78
patch	native	4ms	361	26	4.6m	–
	Valgrind	0.5s	–	–	4.6m	–
	SCOUT	8.0s	356	25	4.6m	85
grep	native	3ms	205	14	1.0m	–
	Valgrind	0.5s	–	–	1.0m	–
	SCOUT	1.8s	208	14	1.0m	73
sed	native	5ms	265	15	12.9m	–
	Valgrind	0.5s	–	–	12.9m	–
	SCOUT	22.1s	266	14	12.9m	77
ls	native	3ms	338	26	0.9m	–
	Valgrind	0.5s	–	–	0.9m	–
	SCOUT	1.6s	338	24	0.9m	73

TABLE 4.5. SCOUT full simulation performance results.

variables errors. The pthread_once test had no main function. Three tests fail when run natively, outside the simulator, within 2 minutes.

There are 293 system calls in Linux Kernel 2.6.7; we currently support 110 of those system calls via marshaling. IBM has contributed a system call test suite to the Linux Test Project with 339 test programs with units tests that test the 110 system calls we support. In those 339 files the total number of passing test conditions is 421 and total number of failing test conditions is 1295.

The sys_gettimeofday test sometimes fails both natively and under SCOUT because the kernel has a bug in the way it interacts with the RTC that causes the clock to go backward by small amounts every so often[2]. The single test of sys_fchown16 does not deterministically pass natively on some systems and we should therefore not expect it to pass under SCOUT either. Four tests of sys_ipc and two tests of sys_fcntl64 fail because our implementation of the futex() system call is incomplete.

Performance Table 4.5 compares SCOUT when concretely simulating the selected programs without jumping to an arbitrary offset. The completion times are real, elapsed time. The SCOUT simulator was an

[2] http://www.gossamer-threads.com/lists/linux/kernel/813344.

optimized version, compiled using "−O3 −fomit−frame−pointer". The specimens are all dynamically linked, 32-bit x86 ELF binaries.

Table 4.5 reports the total number of system calls to demonstrate the fidelity of SCOUT's simulation. Since there is no equivalent tool to SCOUT, we have evaluated it against Valgrind, a powerful, publicly available binary analysis tool. We do not report the system calls Valgrind executes because Valgrind is not natively aware of system calls. Currently, SCOUT shares its standard out with its specimen. As a result, SCOUT does not allow a specimen to close stdout. Some programs respond by retrying to close stdout. Another source of discrepancy is when a program calls execve to start; SCOUT ignores execve because when SCOUT is running, its specimen has already started. SCOUT does not support floating-point, which affected the patch program. In spite of this fact, SCOUT's simulation produced correct output. The unique system call column reports the number of unique system calls made during the run. We did not report this number for Valgrind, again because Valgrind does not natively support collecting such data. Table 4.5 similarly reports the total number of instructions executed, which number in the millions, as well as the unique instructions whose repeated execution generated that total.

Table 4.5 shows that SCOUT is slower than Valgrind, when fully simulating a program. This is not surprising given its greater power. There are three reasons for this slowdown. The first source of slowdown is that SCOUT disassembler is used to build an intermediate representation of each instruction before the instruction can be simulated. Disassembly at some level is necessary in order to simulate instructions, and the level of disassembly that the simulator uses is the same as that which SCOUT uses for all other binary analyses. The second source of slowdown is that dynamic linking in the specimen is resolved as a side-effect of simulation. The simulator loads the specimen's executable into an address space (MemoryMap) just as the host OS would, and then begins simulation. If the main executable has an interpreter, then the interpreter is also loaded and simulation starts in the interpreter. In this way, the simulator is able to simulate dynamically linked executables; it simulates the linker itself (*i.e.*, it simulates the ld−linux.so interpreter). The third source of slowdown is that the specimen system calls is intercepted by hooking the "INT 0x80" or SYSENTER instructions. In either case, the instruction is intercepted and the simulator processes the system call, either by invoking a real system call on the specimen's behalf, or by adjusting the specimen's state to emulate the system call.

4.5.2. Arbitrary Offset. In Chapter 3 we specialized SCOUT to implement semantic clone detection and through that demonstrated some of its capabilities. We created a semantic signature by exercising a function with a randomly generated input and extracting its side effects as output. SCOUT's power comes at a

	instr	nat.	OEP	disasm	**1:1**	**1:10**	**1:100**	**10:1**	**10:10**	**10:100**	**100:1**	**100:10**	**100:100**
ar	42.3	0.02	0.94	14.14	1.14	1.19	1.17	1.62	2.27	2.71	11.40	14.65	54.74
basename	0.2	0.01	0.19	9.79	0.27	0.36	0.88	1.14	1.20	1.53	9.62	11.20	55.52
cal	0.5	0.01	0.27	11.37	0.34	0.33	0.37	1.47	1.10	4.09	12.10	17.15	65.11
diff	2.3	0.02	0.36	12.26	0.44	0.50	0.47	1.32	1.06	6.66	9.35	14.79	68.43
grep	1.0	0.02	0.24	10.66	0.41	0.38	0.40	0.87	1.07	1.33	8.34	17.00	43.04
gzip	5.7	0.01	0.24	9.82	0.33	0.34	0.48	0.73	1.30	2.58	11.23	13.07	25.43
patch	4.6	0.02	0.24	10.27	0.56	0.60	0.52	1.58	1.50	3.22	11.25	20.47	40.34
rmdir	0.2	0.01	0.24	9.83	0.36	0.37	0.54	0.90	1.33	10.29	10.89	14.16	91.00
sed	12.9	0.02	0.24	9.90	0.75	1.72	12.13	1.47	3.26	13.44	10.32	24.45	117.66
tail	0.3	0.01	0.24	10.00	0.35	0.36	0.40	0.87	1.97	8.00	7.26	17.85	34.09

TABLE 4.6. All units are in seconds except for instructions executed (instr), which is in millions. The program is interpreted at an arbitrary instruction N times, and instantiated M times for each offset (N:M).

cost: it is a heavy-weight simulator. Here we inspect the performance properties of the general tool and show how its ability to begin execution at an arbitrary offset reclaims lost performance. In particular, we show how quickly SCOUT can jump to and analyze randomly chosen instructions in 10 commonly used UNIX utilities. To begin from an arbitrary offset, SCOUT must concretely execute the specimen up to a certain point to resolve dynamic linking. Thus, it parses the ELF file to find the address of main and stops when it is reached. By executing to main, SCOUT allows the dynamic linker to run, giving us more information about the executable.

To differentiate pointer and nonpointers, we observe that a pointer dereference has two parts: first we read memory to obtain the value of the pointer, then we use that value as an address from which to perform another read. Thus, we take two passes: In the first pass, SCOUT identifies reads that use the data segment register, disregarding reads that fetch instructions or access the stack. For each such read, SCOUT marks the instructions that defined the address, via a def-use analysis. When two control flow paths merge back into one, SCOUT merges the two sets of defining instructions. In the second pass, SCOUT presumes that, when an instruction marked in the first pass reads from memory, it is obtaining the address of a pointer, which SCOUT then stores. This mechanism is imprecise — an instruction may actually read a value to compute the address of another memory read, which is why the current approach determines that "i" is a pointer in the C expression "a[i]" and "$*$(a+i)".

To lazily instantiate memory for the remaining variables, which are nonpointers, SCOUT solves their associated symbolic expression to generate a random value that reflects what SCOUT was able to learn and encode into that variable's symbolic expression up to that point. SCOUT then overwrites that location with that value so that we get the same value if we do another read from the same address later in the current run.

Table 4.6 shows the elapsed time in seconds to apply this algorithm to realize a lazily instantiated memory in order to interpret an arbitrary binary fragment and to produce input-output pairs from which a programmer can infer that fragment's semantics. For each program, we select N random addresses chosen uniformly at random and, for each address, we interpret from that address M times to compute input-output pairs for each nonpointer storage location.

(1) Load the specimen executable
(2) Simulate the executable up to its OEP to dynamic link
(3) Disassemble the entire process to obtain candidate instructions
(4) Repeat N times:
    (a) Choose a candidate instruction
    (b) Run a pointer-type analysis at that instruction
    (c) Repeat M times and run a MemoryOracle analysis at that instruction
(5) Terminate simulation

Note that there's a lot of variability in how long steps 4.2-4.3.1 take for various candidate instructions chosen by step 4.1. And since both the inner and outer loop are generating random numbers, changing the number of inner loop iterations causes different instructions to be chosen by step 4.1. Therefore, the Analysis-A numbers are not always monotonically increasing as one goes down and/or right in the table.

4.6. Related Work

To our knowledge, SCOUT is the only mixed analysis framework that supports the analysis of binary fragments from any offset. Mixed interpretation allows SCOUT to simultaneously execute multiple, interacting analyses; SCOUT's extensible abstract storage model is also unique and enables SCOUT to begin execution starting at arbitrary offsets. The rest of this section surveys related work, which we classify into the following categories.

Virtual Machine SCOUT is an extensible, per-program virtual machine. There is a wide array of existing virtual machines that support concrete interpretation of binaries. Examples are Bochs [55], Embra [91], QEMU [13], SPIM [80] and PTLsim [93]. However, they all operate on concrete storage, while SCOUT operates on abstract storage and supports concrete, symbolic, and abstract interpretations.

Binary Instrumentation Pin [62], Strata [75, 81], DynInst [67] and Valgrind [61] are runtime instrumentation tools that inject instructions into the instruction stream to perform binary analysis. They are "tethered"

to a concrete execution trace, which means that they achieve scalability by exploring a program's CFG in the neighborhood of a concrete trace, although approaches exist that can perturb the program state along the trace to explore more of the program's state space. PinOS [16] can instrument operating systems and unify user/kernel-mode tracers. It is built upon Xen [10] and provides similar functionality as Valgrind. In comparison to these tools, SCOUT is a per-program virtual machine. Runtime instrumentation approaches do not support abstract storage and thus cannot interpret binaries abstractly or symbolically. Neither can these tools interpret code from arbitrary offsets.

Symbolic Binary Analysis BitBlaze [78], built on top of Valgrind, explores the program execution space around a trace. It is still tethered to the trace and does not support binary analyses of code fragments from arbitrary offsets. Instead, it extracts and changes the program state from a trace to explore the program's additional state space. S^2E [22] is similar to BitBlaze, but adds a more accurate hardware model. S^2E is the first tool that handles all aspects of hardware communication which SCOUT delegates to the host OS. S^2E translates between a concrete representation of a program in QEMU and a symbolic representation in KLEE [18], whereas SCOUT directly supports symbolic execution of instructions. SAGE [36] is a symbolic execution engine for binaries, developed and used internally at Microsoft. Like BitBlaze, it is also tethered to and operates over concrete traces.

Systematic path exploration techniques for source code, such as DART [37], CUTE [76], SJPF [65] and EXE [19], provide the foundation for the above tools. The basic idea is to synergistically combine concrete and symbolic execution to improve test coverage.

Tools that symbolic execute source code either have to model the services provided by system calls or invoke the system calls directly. For instance, modeling the file system has enabled KLEE to test UNIX utilities without invoking the real filesystem [18]. However, creating models is a labor-intensive and error-prone process and researchers has reported spending several person-years writing a model for the kernel/driver interface of a modern OS [8]. Future work will apply SCOUT to the task of simulating a kernel; success here would automatically generate models for external functions including system calls. We currently only generate models for external functions outside the kernel.

Static Binary Analysis As SCOUT has a disassembler component, it is also related to disassemblers and PDG-based tools that enable the static analysis of binaries. Notable examples include CodeSurferx86 [7, 68] and IDAPro [42]. These tools typically lack a complete execution model whereas SCOUT leverages its ability to interpret the code to iteratively refine its disassembly.

4.7. Conclusion

In this chapter, we have presented SCOUT, a pluggable, mixed binary analysis framework. Its novel combination of features allows it to start its analysis in different modes of execution from an arbitrary offset. This capability opens the door to new application domains in the binary analysis arena, such as unit testing of third-party binaries. We believe that SCOUT's power, extensibility, and ease of use will enable the design and development of novel and practical binary analysis tools.

CHAPTER 5

Summary

This dissertation has introduced novel techniques for syntactic and semantic detection of code similarity in binaries. The algorithms and tools we have presented are practical and can automatically find similar code.

This dissertation has reported on work to support syntactic code similarity detection on binaries and address the scale of evaluation required for its broad use in the detection of general properties of commercial off the shelf software. We have shown that although syntactic code similarity detection works well in a non-adversarial setting, compiler optimizations usually defeat purely syntactic attempts to detect code similarity in binaries. In this setting neither semantic approaches, which reports too many false positives, nor syntactic approaches, which have too many false negatives, can effectively solve the problem alone. To overcome this challenge we combine two previously distinct lines of research, syntactic and semantic similarity detection, into a hybrid technique that relies on one approach to combat the weakness of the other. By applying the hybrid semantic and syntactic algorithm to detect code theft we have shown that it can pierce the veil of compiler optimizations and source code obfuscations, and that our hybrid measure is unaffected by such transformations and moreover unaffected by source code obfuscations.

To implement our tool for hybrid syntactic and semantic similarity detection,SLEUTH, we needed the capability to interpret a binary from an arbitrary offset. Current binary analysis tools either analyze a decompiled form without implementing a representation of the execution model of the target machine, or concretely execute a trace from program entry to exit that covers the code fragment. We have presented a framework and tool, SCOUT, with the unique capability of executing a program concretely or abstractly, starting from any offset. Thus, SCOUT can semantically analyze program fragments or functions without a trace that covers the function. SLEUTH is a personality of SCOUT, which it extends and customizes. SLEUTH is limited to concrete execution.

SCOUT's powerful combination of features allows it to begin its analysis in different modes of execution at an arbitrary offset. The concrete execution capability applied in Chapter 3 is a limited demonstration of its capabilities. SCOUT is alone in supporting abstract interpretation of binaries. Abstract interpretation is a powerful framework for static program analysis. This presents a large opportunity in future work to

create abstract semantic signatures, addressing the function coverage problem in concrete execution and state explosion problem of symbolic execution from program entry to program exit.

SCOUT's capabilities also open the door to new application possibilities in binary analysis, including but not limited to static testing, refactoring, unit testing of third party binaries and intellectual property theft detection. SCOUT's powerful capabilities, together with its extensibility and ease of use, will further the development and design of new tools for binary analysis.

Bibliography

[1] IDA Pro disassembler. http://www.datarescue.com.

[2] JPlag. http://www.jplag.de.

[3] AMD64: Architecture programmer's manual volume 3: General purpose and system instructions. http://developer.amd.com/documentation/guides/pages/default.aspx#chipset.

[4] A. Andoni and P. Indyk. E2LSH: Exact Euclidean Locality-Sensitive Hashing. Web: http://www.mit.edu/~andoni/LSH/, 2004.

[5] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[6] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 1997.

[7] G. Balakrishnan. *WYSINWYX: what you see is not what you execute*. PhD thesis, Madison, WI, USA, 2007. AAI3278779.

[8] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys*, 2006.

[9] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The BINCOA framework for binary code analysis. In *CAV*, 2011.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[11] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. In *ESEC/FSE-13*, pages 156–165, 2005.

[12] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS®: Program transformations for practical scalable software evolution. In *ICSE*, pages 625–634, 2004.

[13] F. Bellard. QEMU, a fast and portable dynamic translator. In *ATEC*, 2005.

[14] D. Brumley and J. Newsome. Alias analysis for assembly. Technical Report CMU-CS-06-180[R], Carnegie Mellon University, Oct 2006.

[15] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control flow graph matching. In *DIMVA*, 2006.

[16] P. P. Bungale and C.-K. Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *VEE*, 2007.

[17] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. Technical Report UCB/EECS-2009-133, EECS Department, University of California, Berkeley, 2009.

[18] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[19] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *CCS*, 2006.

[20] S. Carter, R. Frank, and D. Tansley. Clone detection in telecommunications software systems: A neural net approach. In *IWANNT*, 1993.

[21] S. Chaki, C. Cohen, and A. Gurfinkel. Supervised learning for provenance-similarity of binaries. In *KDD*, 2011.

[22] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 46:265–278, March 2011.

[23] R. Chou, L. H. Huffman, R. Fu, A. K. Smits, and P. T. Korthuis. Screening for HIV: A review of the evidence for the U.S. preventive services task force. *Ann. Intern. Med.*, 2005.

[24] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *USENIX Security*, 2003.

[25] M. Christodorescu and S. Jha. Testing malware detectors. In *ISSTA*, 2004.

[26] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. In *S&P*, 2005.

[27] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114:494–509, 1993.

[28] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, 1977.

[29] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In *Symposium on Computational Geometry*, 2004.

[30] T. Dullien, R. Rolles, and R. universitaet Bochum. Graph-based comparison of executable objects. In *University of Technology in Florida*, 2005.

[31] B. Dutertre and L. M. D. Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification*, pages 81–94, 2006.

[32] H. Flake. Structural comparison of executable objects. In *DIMVA*, 2004.

[33] M. Gabel, L. Jiang, and Z. Su. Scalable Detection of Semantic Clones. In *ICSE*, pages 321–330, 2008.

[34] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *ICICS*, 2008.

[35] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Very Large Data Bases*, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[36] P. Godefroid, P. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating software testing using program analysis. *IEEE Softw.*, 25:30–37, September 2008.

[37] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[38] A. Hemel. The GPL compliance engineering guide. http://www.loohuis-consulting.nl/downloads/compliance-manual.pdf.

[39] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra. Finding Software License Violations Through Binary Code Clone Detection. In *MSR*, 2011.

[40] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *CCS*, 2009.

[41] L. Humphrey, M. Helfand, B. Chan, and S. Woolf. Breast cancer screening: a summary of the evidence for the U.S. preventive services task force. *Ann. Intern. Med.*, 2002.

[42] IDAPro: a disassembler. http://www.idapro.com/.

[43] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *ACM Symposium on Theory of Computing*, 1998.

[44] Intel IA-32: Architectures software developer's manuals. http://www.intel.com/products/processor/manuals/.

[45] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu. Value-based program characterization and its application to software plagiarism detection. In *ICSE*, 2011.

[46] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE*, 2007.

[47] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA*, 2009.

[48] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[49] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A Search Engine for Binary Code. In *MSR*, 2013.

[50] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *CAV*, 2008.

[51] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[52] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, 2001.

[53] C. Kruegel, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, 2005.

[54] A. G. Lalkhen. Clinical tests: sensitivity and specificity. *Contin. Educ. Anaesth. Crit. Care*, 2008.

[55] K. P. Lawton. Bochs: A portable PC Emulator for Unix/X. *Linux J.*, 1996, September 1996.

[56] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In *OSDI*, 2004.

[57] LTP: Linux test project. http://ltp.sourceforge.net/.

[58] M. Madou, L. Van Put, and K. De Bosschere. LOCO: An Interactive code (De)Obfuscation Tool. In *PERM*, 2006.

[59] J. McCarthy. Towards a mathematical science of computation. In *In Proceedings of the IFIP Congress*, pages 21–28, 1962.

[60] G. Myles and C. Collberg. K-gram based software birthmarks. In *SAC*, 2005.

[61] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[62] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *MICRO'37*, 2004.

[63] I. V. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *Usenix Security*, 2011.

[64] POSIX.1-2008: The Open Group Base Specifications Issue 7. http://pubs.opengroup.org/onlinepubs/9699919799/.

[65] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, 2008.

[66] D. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[67] G. Ravipati, A. Bernat, B. P. Miller, and J. K. Hollingsworth. Towards the deconstruction of Dyninst. Technical report, 2006. http://www.dyninst.org/.

[68] T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A next-generation platform for analyzing executables. In *In APLAS*, pages 212–229, 2005.

[69] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *ISSTA*, 2009.

[70] M. Schäfer, T. Ekman, and O. de Moor. Challenge proposal: verification of refactorings. In *PLPV*, 2009.

[71] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *Management of Data*, pages 76–85, 2003.

[72] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, Aug. 2003.

[73] A. Schulman. Finding binary clones with opstrings and function digests. *Doctor Dobb's J*, 30(9):64–70, 2005.

[74] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, Oct. 1980.

[75] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO*, 2003.

[76] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30:263–272, September 2005.

[77] G. Shakhnarovich, T. Darrell, and P. Indyk. *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice (Neural Information Processing)*. The MIT Press, 2006.

[78] 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 *ICISS*, 2008.

[79] Sensitivity and Specificity. http://en.wikipedia.org/wiki/Sensitivity_and_specificity.

[80] SPIM. http://pages.cs.wisc.edu/~larus/spim.html.

[81] Strata. http://dependability.cs.virginia.edu/info/Strata#Papers.

[82] Stunnix source code obfuscator. http://www.stunnix.com/.

[83] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. *SIGPLAN Not.*, Oct. 2000.

[84] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. Directed proof generation for machine code. In *CAV*, 2010.

[85] W. M. Thomas, A. Delis, and V. R. Basili. An analysis of errors in a reuse-oriented development environment. *J. Syst. Softw.*, 38(3), 1997.

[86] V. Wahler, D. Seipel, J. W. von Gudenberg, and G. Fischer. Clone Detection in Source Code by Frequent Itemset Techniques. In *Source Code Analysis and Manipulation*, pages 128–135, 2004.

[87] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Behavior based software theft detection. In *CCS*, 2009.

[88] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Detecting software theft via system call based birthmarks. In *ACSAC*, 2009.

[89] E. J. Weyuker. Evaluating software complexity measures. *IEEE Trans. Softw. Eng.*, 14(9):1357–1365, 1988.

[90] Wikipedia: X86 instructions. en.wikipedia.org/wiki/X86_instruction_listings.

[91] E. Witchel and M. Rosenblum. Embra: fast and flexible machine simulation. *SIGMETRICS Perform. Eval. Rev.*, 24:68–79, May 1996.

[92] H. Yin, D. X. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS*, 2007.

[93] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS*, 2007.

[94] R. Zippel. Probabilistic algorithms for sparse polynomials. In *EUROSM*, volume 72. Springer Berlin Heidelberg, 1979.