2013 IEEE 37th Annual Computer Software and Applications Conference

Exposé: Discovering Potential Binary Code Re-Use

Beng Heng Ng, Atul Prakash
University of Michigan
Department of Electrical Engineering and Computer Science
Ann Arbor, Michigan 48105, USA
{bengheng, aprakash}@eecs.umich.edu

***Abstract*—The use of third-party libraries in deployed applications can potentially put an organization's intellectual property at risk due to licensing restrictions requiring disclosure or distribution of the resulting software. Binary applications that are statically linked to buggy version(s) of a library can also provide malware with entry points into an organization. While many organizations have policies to restrict the use of third-party software in applications, determining whether an application uses a restricted library can be difficult when it is distributed as binary code. Compiler optimizations, function inlining, and lack of symbols in binary code make the task challenging for automated techniques. On the other hand, semantic analysis techniques are relatively slow. Given a library and a set of binary applications, we propose Exposé, a tool that combines symbolic execution using a theorem prover, and function-level syntactic matching techniques to achieve both performance and high quality rankings of applications. Higher rankings indicate a higher likelihood of re-using the library's code.**

Exposé ranked applications that used two libraries at or near the top, out of 2,927 and 128 applications respectively. Exposé detected one application that was not detected by another scanner to use some functions in one of the libraries. In addition, Exposé ranked applications correctly for different versions of a library, and when different compiler options were used. Exposé analyzed 97.68% and 99.48% of the applications within five and 10 minutes respectively.

***Keywords*—binary code re-use; semantic analysis; syntactic analysis**

I. Introduction

Code re-use in binaries can lead to security implications including detecting software license violations, intellectual property theft [1], and vulnerabilities resulting from re-using vulnerable code [2], [3]. The problem of identifying code re-use has been studied extensively for the case of source code. It includes works such as ReDeBug [3], DECKARD [4], CCFinder [5], and CP-Miner [6]. However, only few works, such as BinHunt [7], BitShred [8], and SMIT [9], have been proposed for the case of detecting binary code re-use. Reps et al. advocate analyzing executable binary code as it is the authoritative source of information for an application, and note that there are at least three varieties of binary code analysis problems: (i) source code and binary are available, (ii) binary with symbol/debugging information is available, and (iii) binary without symbol/debugging information (stripped binary) [10]. According to Reps et al., the third type of problem is the most challenging, and is the common situation when one uses off-the-shelf applications. We examine the third type of problem for code re-use, and aim to address the following challenges in a scalable manner.

1) **Function inlining** Compilers often perform inlining on small functions to eliminate function call overheads.
2) **Lack of Symbol Information** Symbols, which can be helpful in matching functions, are commonly absent. Our analysis shows that at least 80% of the applications lack symbols.
3) **Code mutation due to compiler options** Different compiler options can lead to syntactically different but semantically equivalent code being emitted.

Organizations go to great lengths to prevent leakage of their software intellectual property. But, significant risks exist due to channels that are difficult to monitor and control. One channel occurs when a developer within an organization inadvertently uses a library with a license that would later require the organization to openly release the software that makes use of the library. One author ran into this issue when using Berkeley DB, thinking that it was under Berkeley license and thus acceptable to use on a project with a company without a fee. It turned out that while the early versions of Berkeley DB were indeed under the Berkeley license, more recent versions were significantly more restrictive, and could have put the company's software at risk with the free version of the license. In another case, researchers found an internet filtering software using libraries for image recognition from OpenCVS without including a copy of its BSD license text as stipulated by their authors [11].

Another channel that leads to security risks is the use of applications that are statically linked to an unpatched version of a library. Software library usage leads to a situation where bugs and vulnerabilities in the libraries can be inherited by the applications. Often, the associations between a library and statically-linked applications are not tracked (developers may know it, but users usually do not). Examples of such binaries are (i) customized in-house applications that are not actively maintained, (ii) third-party software, and (iii) pre-installed binaries that come with the operating systems. Over time, bugs and vulnerabilities may be discovered and patched in the libraries. However, they can continue to remain in such applications for a while. Ng et al. termed this phenomenon as latent vulnerabilities [2]. The problem is also studied by Jang et al. in their work on ReDeBug [3], albeit using source code (our focus is on binary code), for uncovering such vulnerabilities.

Towards identifying binary code re-use, we propose Exposé for identifying potential code re-use between a library and a set of applications. This does not necessarily mean that identified applications derive the re-used code from the library directly. Besides statically linking with the library, a developer could have compiled the application containing source code for a library. Abstractly, in identifying library re-use, we want to identify the set of functions from the library that are used within an application. Unfortunately, this can be a tedious task, even with manual analysis using disassemblers and analysis tools like

Ida Pro. If an organization wishes to do this for a large number of applications, the task can be overwhelming.

The main contributions of our work include the following.

- We identify function inlining, lack of symbol information, and code mutation due to varying compiler options as significant challenges for detecting code re-use in binaries.
- To overcome these challenges while ensuring scalability, we propose a multi-phase technique, with the first phase being a fast pre-filtering step to identify a small set of relevant functions in the application that are good candidates for doing a match with the library functions. Subsequent phases do a more thorough matching to ensure a high-quality match. For the first phase (fast), we identified four attributes – number of input parameters, out-degree, function size, and cyclomatic complexity – that determine a set of candidate function pairs (between a library and an application) for a subsequent phase (slower) of semantic matching using a theorem prover. We also use syntactic techniques for matching functions that are not amenable to semantic matching. We propose a method to compute the distance scores used for ranking applications in the order of likelihood that they are using the library.
- We implemented Exposé to evaluate our techniques. Exposé successfully ranked an application known to use a library above a set of 128 other applications known not to use the same library. Using another test library, Exposé ranked 10 out of 2,927 applications amongst the top 11 positions. We verified the correctness with a signature scanner. On manual investigation, the single application, which was not detected by the scanner to use the library, was identified as most likely using some functions from the library.
- Exposé was able to distinguish applications using the different versions of a library, as well as variants compiled using different compiler options.
- Exposé analyzed 97.68% and 99.48% of the applications within five minutes and 10 minutes respectively.

In Section II, we discuss the scope and limitations of the work. In Section III, we present existing techniques in software similarity research and how our work distinguishes from them. In Section IV, we detail our approach. In Section V, we provide experimental results and evaluation of our technique. And finally, we conclude in Section VI.

II. Assumptions and Scope

In this paper, when we say that an application "uses" a library, we mean that the application is either statically linked to the library, or has been compiled together with the library's source code. Our work focuses on examining these more challenging cases rather than dynamically linked libraries, where the names of the shared libraries typically provide enough information to correctly identify them; moreover, the names of exported functions can be easily extracted and leveraged for identification purpose. Also, updating a dynamically linked library will automatically update the applications that dynamically link to it. However, static linkage of code from libraries continues to be common. An advantage of static linkage is that the library does not have to be distributed with the code and fewer assumptions need to be made about the availability of the library.

We acknowledge the challenge in providing an efficient and effective solution for the code re-use problem. Thus, we focus our efforts in finding a solution that is *practical*. By practical, we mean that the technique should rank candidate applications with true positives in the highest ranks within a reasonable amount of time. The sorted binaries can then be prioritized for further analysis for the presence of specific functions of interests using a combination of manual and more detailed techniques that are typically less scalable.

TABLE I: Percentage of binaries without symbols in various Linux distributions.

Distributions	Total # of Files	# without Symbols	
		(#)	(%)
Ubuntu 10.04	4268	4254	99.672
Fedora 13	4355	4080	93.685
DVL 1.5	13325	10760	80.750
Debian 5.0.4	4077	4039	99.068
Mandriva Free 2010	5610	5603	99.875

To further complicate the problem of identifying library re-use, many applications are optimized so that they lack symbol data. Symbol data could have been used to identify names of functions that are used within an application, which could be checked against the names of functions in a library. We ran a script that checks for presence of symbols against the binaries from several Linux distributions. As shown in Table I, at least 80% of binaries on the systems do not have symbols. This is not surprising as symbols consume unnecessary disk space on production systems. We therefore assume the absence of symbols in this work. Of course, symbols can be trivially leveraged to improve our results in practice. Implicitly, tools leveraging symbols, such as `objdump`, will not suffice for most cases.

We assume a benign library usage environment in which functions from the libraries or an entire library is used to build applications. A developer is assumed not to be attempting to obfuscate the use of the functions in the library. Existing techniques to de-obfuscate the binary, such as [12] and [13], can be applied if necessary. Moreover, the problem is significant enough even without obfuscated library re-use.

Lastly, we would like the matching to be relatively insensitive to different compiler options. For example, we do not make any assumptions about compiler optimizations that result in basic block re-ordering or function inlining.

III. Related Work

This work falls under the category of software similarity research. Collberg and Nagra sub-categorized this field based on four applications: software birthmarking, software forensics, plagiarism detection, and clone detection [14]. The objective of our work is most similar to that of birthmark detection.

A. Syntactic Approaches

1) n-gram: Many of the works in birthmark detection are designed for interpreted languages, typically Java (when compiled as Java bytecodes) [15], [16], [17], [18], [19]. There are other works that, while not specific to Java bytecodes, use them for evaluation, such as the work by Myles et al. [20]. The number of user-configurable options affecting the output bytecodes during compilation of Java source code is limited. For example, the original Java compiler `javac` does not provide any option that affects how the bytecodes may be generated. This implies that given the Java source code, compilations undertaken by different developers would likely yield highly

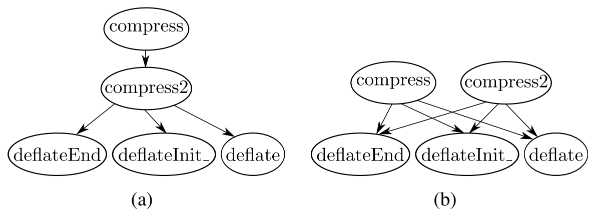

Fig. 1: (a) Partial function call graph of a shared library. (b) Partial function call graph of an executable statically linked with the shared library showing function inlining.

similar Java bytecodes. In contrast, compilers for compiled languages such as C/C++ offer users different options, such as optimization levels and function inlining threshold, which introduce variations in the output native code. Since Exposé models the sequence of opcodes using n-gram, the variations can give rise to both false positives and false negatives. The effect of false negatives is generally smoothened out by true positives. False positives tend to be more problematic, which we mitigate using semantic analysis.

Outside the domain of birthmark detection, several proposals on using n-gram for malware detection have been proposed previously [6], [8], [21], [22], [23], [24], [25]. These work generate file signatures using n-gram, which are used for classification. Exposé differs from these works in that we apply n-gram at the function level, which provides us more precision without suffering from the consequences of basic block reordering if we had used the basic block abstraction level. To the best of our knowledge, we believe our work is the first to perform in-depth analysis of applying n-gram at the function level for solving the code re-use problem in binaries.

2) Approximate Graph Matching: Another approach is to identify syntactically similar binaries by using approximate graph matching on the function call graphs. Hu et al. proposes SMIT, a scalable malware database management system for determining if a new malware is similar to one that is previously observed [9]. Hu et al. modified the Munkres algorithm (also called Hungarian algorithm [26]) under the assumption that if two nodes match, then their neighbors are also likely to match. However, if this assumption fails, such as when function inlining occurs, the error may propagate to all neighbors, thus misleading the Munkres algorithm to produce an incorrectly biased set of assignments. Function inlining typically happens as part of the compiler's optimization process, which eliminates function calls, thereby improving runtime performance. Figure 1 shows part of the function call graphs for two binaries. Figure 1(a) shows the function call graph of a library, while Figure 1(b) shows the graph for an application that is statically linked with the same library. In Figure 1(a), `compress` calls `compress2`, which in turn calls three other functions. However after being statically linked with an application, `compress2` is inlined into `compress`. Exposé attempts to mitigate the effects of function inlining by allowing a library function to match with both another application function and its caller.

B. Semantic Approaches

1) API-based: In addition to n-gram, other possible birthmarks include API sequence, call structures, and frequencies. Choi et al. propose extracting a static birthmark based on the API call structure [27] while Tamada et al. extract the API information dynamically [28]. Wang et al. propose using System Call Dependency Graph (SCDG), also extracted dynamically, as birthmark [29]. The use of API and system call information is useful if such information is available and unique. However, for our purpose, this information may not necessarily be available. The symbols may be stripped and thus possibly removing the function names. Also, for self-contained executables that do not make system calls, such as compression libraries, these features cannot be applied.

2) Symbolic Execution: `BinHunt`, a work by Gao et al., attempts to find semantic differences between binaries using symbolic execution and theorem proving [7]. The authors propose finding semantic differences instead of syntactic differences. They argue that semantic differences are more suitable for reflecting changes in program functionalities. They attempt to compute the matching strength between basic blocks within two functions across two binaries using symbolic execution to extract a set of constraints that are then submitted to the theorem prover. A backtracking algorithm, guided by the matching strength, is then applied to identify similarities between the control flow graphs of the functions. In their first case study, it required about an hour to complete the examination of several basic blocks. The authors explained that the long computation time is due to syntactic differences between the functions. In their second case study, analyzing two similar binaries of about 41,000 instructions took approximately 30 minutes. While `BinHunt` is able to identify semantic differences with high accuracy, it is less practical for analyzing huge number of files. Exposé aims to quickly rank a set of executables with high true positive rate. `BinHunt` may then be applied on the smaller set of files to determine if they are semantically identical with a vulnerable library file.

C. Other Techniques

It may seem immediate that the state-of-the-art techniques for identifying obfuscated malware could be trivially applied for our problem domain. We first examine static analysis approaches to malware detection. Moser et al. show that static analysis, which encompasses syntactic analysis, is ineffective in identifying obfuscated malware and propose combining static with dynamic analysis [30]. Malware static analysis generally identifies unique syntaxes such as section names and the legitimacy of the entry point for the program for hints. Other hints include entry point in the thread local storage section, shared libraries with no exported functions, low import function count, and modified import function table [31]. These techniques were effective until malware authors removed these hints. In our context, the absence of such hints renders the techniques ineffective.

Recent works in identifying obfuscated malware have focused primarily on semantic analysis [32], [33]. While we also use semantic analysis as a building block, it is not entirely sufficient. One subtle difference between malware analysis and Exposé is that the former identifies equivalences between binaries while the latter focuses on discovering code re-use. In malware analysis, it is typically assumed that two functionally equivalent binaries do not contain extraneous functions so as to minimize file size. The function call graph usually remains mostly similar and has a smaller order. On the other hand, in addition to library functions, applications mostly comprise other functions, thus introducing substantial noise. Functions in the library may

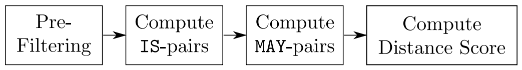

Fig. 2: Approach overview.

not be linked into the application if they are not called by the application, resulting in modified function call graphs. These interferences make our work an interesting and a challenging one. A second subtle difference is that semantic analysis may fail due the complexity of the functions and does not scale well if a large number of functions in a library need to be compared with functions in a large number of applications.

IV. Approach

The inputs to Exposé are two sets of disassembled functions, F and G, that are respectively derived from a library and an application. The library functions in F are extracted from a single object file generated by linking all the object files. The functions in G are simply extracted by analyzing the application binary. In both cases, extracting the function call graphs is trivial since the linker resolves all references. The goal is to find the set of functions in F that are likely to be in G. From the results of the match, Exposé attempts to compute a matching score that can be used to rank the results from a large number of applications for a given library. A smaller score corresponds to more similarities.

Figure 2 shows an overview of the different phases for our approach. The pre-filtering phase prepares a set of candidate function pairs for computing semantic equivalence by removing improbable function pairs and functions of no interest to us. Next, we compute the `IS`-pairs to identify equivalent functions. For the remaining functions, we compute the `MAY`-pairs using syntactic techniques. Exposé then computes a distance score that summarizes the pairings by considering the caller/callee relationships.

Procedure 1 provides an algorithmic overview of the various phases for our technique. In the following sub-sections, we discuss the details of each phase.

Procedure 1 Compute distance score between two sets of functions F and G.

```
Input: Disassembled functions $F = \{f_1, f_2, \ldots, f_m\}$
Input: Disassembled functions $G = \{g_1, g_2, \ldots, g_n\}$
Output: Distance score $s$
  IS-Pairs = $\emptyset$
  MAY-Pairs = $\emptyset$

  {STEP 1: Populate IS-pairs with function pairs that are semantically
  equivalent and MAY-pairs with pairs that are either not tested for semantic
  equivalence or test does not return true.}
  for $i = 1 \to m$ do
    $A_i$ = ExtractAttributes( $f_i$ )
    for $j = 1 \to n$ do
      $B_i$ = ExtractAttributes( $g_j$ )
      if MeetCriteria( $f_i$, $A_i$, $g_j$, $B_j$ ) $\wedge$
      IsSemanticallyEquivalent( $f_i$, $g_j$ ) then
        {STEP 1a: Test for semantic equivalence between $f_i$ and $g_j$.}
        IS-Pairs = IS-Pairs $\cup$ ($f_i$, $g_j$)
      else
        {STEP 1b: Add ($f_i$, $g_j$) to MAY-Pairs. The MAY-Pairs data
        structure keeps at most 5 $g_j$'s per $f_i$ with the smallest cosine
        distance from $f_i$.}
        MAY-Pairs.add($f_i, g_j$)
      end if
    end for
  end for

  {STEP 2: Compute localdist score for each MAY-pair.}
  for all $(f_i, g_j, c_{i,j}) \in$ MAY-Pairs do
    $localdist[f_i][g_j]$ = CalcLocalScore( $(f_i, g_j, c_{i,j})$ )
  end for

  {STEP 3: Eliminate inconsistent MAY-pairs. }
  $C = \emptyset$
  for $i = 1 \to m$ do
    $u = RowMin(localdist, i)$
    for $j = 1 \to n$ do
      $v = ColMin(localdist, j)$
      if $u = v$ then
        $C = C \cup \{(f_i, g_j)\}$
      end if
    end for
  end for

  {STEP 4: Form function groups.}
  $H$ = FunctionGrouping( C )

  {STEP 5: Find average distance of scaled localdist scores and divide
  by number of IS-pairs.}
  $s = 0$
  for all $(f_i, g_j) \in C$ do
    $s = s + localdist[f_i][g_j]/SizeOfGroup(H, f_i)$
  end for
  $s = s/SizeOf(C)$
  $s = s/$SizeOf(IS-Pairs)
  return s
```

A. Pre-Filtering

Symbolic execution has the advantage of being sound under typical variations in instruction opcodes and optimizations that may be introduced by compilers. But, comparing two functions via symbolic execution can have a scalability problem, usually as a consequence of having many possible paths through the functions. The pre-filtering phase selects candidate function pairs by excluding loader support functions and improbable function pairs to avoid unnecessary symbolic executions, which are computationally intensive.

Loops also require special handling to limit the search space. These problems are widely acknowledged and are currently an area of intense research [34], [35], [36], [37]. To simplify the problem, we only consider the first iteration of each loop.

1) Excluding Loader Support Functions: Given a binary, identifying the functions is basically straightforward, except for a few nuances. During compilation, compilers insert loader support functions to provide support for loading, executing, and terminating the application. However, varying functions are inserted by the compiler depending on the compilation options. For example, stubs such as `__i686.get_pc_thunk.bx` or `__i686.get_pc_thunk.cx` are inserted when the `fPIC` option is specified. To avoid misleading matches with these functions, we exclude them from our pairing algorithm. We list these functions in Table II for the `GCC` compiler. Table II shows the common functions that we have observed and may not be exhaustive. Procedure 1 assumes that these functions are already excluded.

2) Excluding Improbable Function Pairs: Ideally, we would like to test all functions in F with functions in G for semantic equivalence using symbolic execution. But, this would not be a scalable approach. Towards achieving a balance between scalability and correctness, Exposé uses the following criteria based on attributes that are easy to compute for quickly filtering out non-probable function pairs and selecting suitable candidate pairs to test for semantic equivalence.

`__do_global_ctors`	`_fini_array`
`__do_global_dtors`	`__fini_array_start`
`__do_global_ctors_aux`	`__fini_array_end`
`__do_global_dtors_aux`	`frame_dummy`
`__i686.get_pc_thunk.bx`	`start`
`__i686.get_pc_thunk.cx`	`_start`
`_init_array`	`__libc_csu_fini`
`__init_array_start`	`__libc_csu_init`
`__init_array_end`	

TABLE II: List of common functions excluded.

- Same number of input arguments.
- Same out-degrees.
- Cyclomatic complexity of less than 15.
- Function size of less than 300 bytes.

Two functions are more likely to be equivalent if they have the same number of input arguments and out-degrees. Cyclomatic complexity reflects the number of independent paths [38]. Symbolically executing a function with high cyclomatic complexity quickly degenerates into the path explosion problem. Also, large functions, which usually imply more execution paths, generally lead to disproportionate slowdowns when computing for semantic equivalences. We chose to perform symbolic execution on function pairs with cyclomatic complexity less than 15 and function sizes less than 300 bytes. 455,078 (78.06%) of 582,959 functions we examined satisfy these criteria. In Procedure 1, the sub-procedure `MeetCriteria` is responsible for rejecting improbable function pairs.

B. Computing semantic matches (`IS`-pairs)

If they are semantically equivalent, Exposé adds the pairing to the `IS`-pairs set. We define functions f_i and g_j to form an `IS`-pair if f_i is semantically equivalent to g_j. For determining semantic equivalence, we use the method proposed by King in which a sequence of instructions are executed symbolically by substituting the input variables with symbolic formulas [39].

To perform symbolic execution, each instruction is translated into a set of constraints on the symbolic formulas. Additionally, constraints on the path conditions and input variables are created. We assert that the input variables for both functions are equivalent. We then extract a set of outputs as a consequence of the function executing along each path. A theorem prover is then used to query for the satisfiability of the output equivalences.

To compare paths in the two functions, Exposé uses the STP constraint solver proposed by Ganesh et al. [40]. We modified the translator from the Binary Analysis Platform by Brumley et al. [41] to process x86 instructions and interface directly with STP. We assume that functions use `GCC`'s default calling convention, `cdecl` [42]. In `cdecl`, the calling function pushes function arguments onto the stack from right to left, and the return value is saved in the `EAX` register. We did not include support for other calling conventions as there is currently no efficient method for distinguishing all calling conventions.

We consider two functions to be a matched `IS`-pair if we can find satisfiable output equivalences between the paths in the two functions. On the other hand, if the output equivalences cannot be satisfied, we cannot assert that the two functions are not equivalent. This is because of the simplifications in the matching process to achieve scalability. For example, we assumed that functions called within the functions being matched to have a null behavior to reduce matching cost and avoid inter-procedural symbolic execution. As a result, inlining could cause two matching functions to diverge. For example, suppose f_x calls f_y in the library and the corresponding code for f_x in the application is g_x. We may fail to match f_x with g_x if f_y is inlined with f_x to produce g_x during compilation.

We note that when we find two functions f and g to form an `IS`-pair, they are equivalent only under some abstractions. For example, one abstraction we make is that the behaviors of any functions called by f and g are ignored. We also ignore possible differences in the values of CPU registers at the end of the execution since compiler optimizations can introduce differences in the use of registers. Our hope is that these approximations will turn out to be acceptable on real data sets since the end goal is to narrow down the candidate set of applications to a small number that can be manually analyzed.

It turns out that the lack of a semantic match does not necessarily mean that the functions are semantically different. If a match is found, then the functions are equivalent. But the absence of a match does not prove that the functions are different. For example, semantic matching is sensitive to order of parameters and return values. If we make a wrong assumption about the calling conventions, semantic matching could fail to return a match.

Semantic matching can also return false positives if some outputs or side-effects of the functions, e.g., register values, are important, but not modeled as inputs or outputs. We found instances of those to be rare on practical data sets. We found one instance in our data sets and it was due to a function stub in both the library and the application.

C. Syntactic function matching (`MAY`-pairs)

For the set of library functions for which Exposé is unable to find `IS`-pairs, we adopt a heuristic approach for generating matched function pairs, called `MAY`-pairs based on a similarity measure. False positives can be a problem with such approaches. To help reduce false positives significantly, we factored in both function-level similarity and caller-callee relationships. A `MAY`-pair is a pairing between two functions f_i and g_j such that the following two properties are satisfied.

1) f_i makes a function call to itself if, and only if, g_j makes a function call to itself. That is, f_i is recursive if, and only if, g_j is recursive.
2) The biased cosine distance value for the f_i and g_j pair is amongst the r-th smallest biased cosine distance values for pairings between f_i and all functions in G.

Implementation-wise, we chose r to be five. Thus, for each function in the library, we first identified up to five similar functions in an application using a biased cosine distance.

Identifying function recursion is trivial. Therefore we will focus our discussion on the second property. We compute the cosine distance of the n-gram word frequencies between two functions f_i and g_j. The biggest challenge in computing syntactic distances is in handling syntax variations. Ensuring exactness will result in intolerance of any variation in the syntax. On the other extreme, correctness is compromised if all forms of syntax are tolerated. Thus Exposé uses n-gram in a bid to smoothen slight variations in the syntax. An advantage of using the n-gram based approach is its robustness against code relocations made by the compilers, primarily for the purpose of optimization. Code relocations occur during block re-ordering and function inlining.

It is possible that compilers use instructions that are semantically equivalent but syntactically different. For example, "`mov` eax, 0" is semantically equivalent to "xor eax, eax", but syntactically different. One approach to mitigate such differences is to normalize the instructions such that all instructions that are members of a set of semantically equivalent instructions are replaced with the same instruction. While this is a possible improvement to Exposé, we find that the effects of semantically equivalent but syntactically different instructions are usually smoothened out by syntactically equivalent instructions.

Listing 1 shows the instructions in a basic block of the shared library `libz.so` and Listing 2 shows the instructions for a semantically equivalent basic block. Three of the opcodes differ between the two basic blocks despite having similar mnemonics. Since sequences of instructions having the same mnemonics are likely to be semantically similar, we assign intermediate representation values to similar mnemonics, as shown in the first column of the two listings. Additionally, we do not consider the operands when computing the trigrams since, intuitively, they are highly variable.

Listing 1 Basic block in `libz.so`.

```
IR   Opcode                   Mnemonic
7A   8B 6B 70                 mov     ebp, [rbx+70h]
7A   48 8B BB 80 00 00 00     mov     rdi, [rbx+80h]
D2   85 ED                    test    ebp, ebp
EF   41 0F 49 EC              cmovns  ebp, r12d
D2   48 85 FF                 test    rdi, rdi
55   74 05                    jz      short loc_2CC2
10   E8 96 F0 FF FF           call    _free
```

Listing 2 Semantically equivalent basic block in `modprobe`.

```
IR   Opcode                   Mnemonic
7A   44 8B 63 70              mov     r12d, [rbx+70h]
7A   48 8B BB 80 00 00 00     mov     rdi, [rbx+80h]
D2   45 85 E4                 test    r12d, r12d
EF   44 0F 49 E5              cmovns  r12d, ebp
D2   48 85 FF                 test    rdi, rdi
55   74 05                    jz      short loc_4054F1
10   E8 E7 C0 FF FF           call    _free
```

A larger value of n would require the binaries to have longer common subsequences of opcodes for a good match. This implies that the number of true positives is also reduced for large n, and the algorithm is less tolerant of opcode variations. On the other hand, if n is too low, the number of false positives will be high as short common subsequences are sufficient for the binaries to match. Empirically, we find three to be a good value for n. Thus, Exposé computes trigrams for the opcodes. Using the intermediate representation, the first trigram word in Listing 1 would therefore be (0x7A, 0x7A, 0xD2). Computation of n-grams can be expensive in both space and time. We implemented Nagao et al.'s efficient algorithm that does not require a separate table for storing the n-gram words during computation [43].

1) Eliminating Function Prologues and Epilogues: Function *prologues* and *epilogues* do not contribute any additional information on whether two functions are equivalent. A function prologue is responsible for setting up the stack frame. A function epilogue destroys the stack frame and restores the original value of the base pointer. Table III shows the typical function prologue and two equivalent epilogues on 32-bit machines. We exclude prologues and epilogues from the generation of our n-gram words.

2) Computing Cosine Distance from n-grams: Suppose we let N_x be the vector of trigram word frequencies in x. Then we can define the cosine distance between $f_i \in F$ and $g_j \in G$ as

Prologue	Epilogues	
`push ebp`	pop ebp	leave
`mov ebp,esp`	retn	retn

TABLE III: Typical x86 function prologue and two possible epilogues.

in Equation 1.

$$c_{i,j} = 1 - \frac{N_i \cdot N_j}{|N_i| \cdot |N_j|} \tag{1}$$

Cosine distance computes the cosine of the angle between two vectors of the same dimension. A value of 1 indicates that the two vectors are fully independent, while a value of 0 implies they are exactly the same.

It is possible for two distinct function pairs to have the same cosine distance value. To encourage function pairs that are more likely to be similar and discourage those that are unlikely to be similar, we bias the cosine distance value, $c_{i,j}$, using three rules. Firstly, if f_i and g_j have different out-degrees, increase $c_{i,j}$ by 0.1. Functions with different out-degrees are less likely to be similar. However, it is still possible for such functions to be similar, such as when inlining occurs. Next, if f_i and g_j have the same out-degrees, decrease $c_{i,j}$ by 0.05. Lastly, if f_i and g_j have the same number of non-zero input parameters, decrease $c_{i,j}$ by 0.2. The values used for biasing are derived empirically and we have found them work well in our experiments.

D. Distance Score

Given the results of matches for each function in the library to a corresponding function in the application, Exposé computes a summary score to help rank the results. A match could be generated by semantic equivalence (which is a Boolean value) or by the heuristic approach (which is a value between 0 and 1, with lower-values indicating higher similarity). We now explain how the distance score for the two sets of functions F and G is computed. The procedure comprises three main steps. The first step involves computing the local score by observing the neighbors of the two functions f_i and g_j whose pairing can be found in MAY-pairs. Based on the local scores, the second step identifies candidate function pairs whose functions have corresponding lowest local scores. The third step attempts to group functions in the library that match functions in the application with the same caller/callee relationships.

1) Computing Local Scores: Given the set of MAY-pairs, we want to find the best possible match. Recall that for each function in the library, Exposé has five potential MAY-pairs to functions in an application based on best cosine scores. It seems that we could just run a mincost bipartite matching algorithm, but we found that it can give undesirable results, which eventually cause too many false positives. The problem is that it does not take into account caller-callee relationships and if f erroneously matches with g, the error can propagate because it prevents g from being correctly matched.

To get more robust matching, Exposé computes a local score, $l_{i,j}$, for every MAY-pair (f_i, g_j) as shown in Procedure 2 (in Appendix), factoring in the callers and callees of f_i and g_i. The recursive procedure will lead to better similarity scores for functions with similar callers/callees. To account for function inlining, the procedure allows pairing between f_i and a caller function of g_j when computing the minimum cost assignment. For the same reason, we also allow pairings between a callee

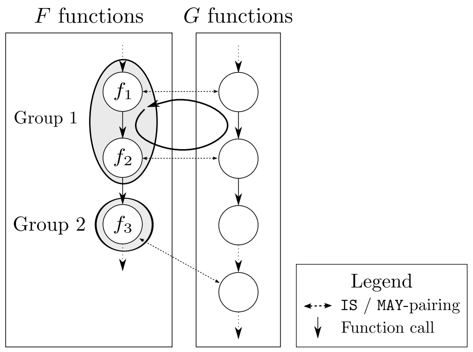

Fig. 3: Function grouping, given a set of matching pairs.

function of f_i with g_j. The Hungarian algorithm is used to find the best localized mapping between the caller/callee functions of f_i and g_j.

2) Inconsistent `MAY`*-pairs Elimination:* To help find a high-quality bipartite matching from the pairs, Exposé identifies and removes *inconsistent* `MAY`-pairs after computing the local scores for all `MAY`-pairs. We consider the pairing between two functions f_i and g_j to be consistent only if the minimum local score for f_i corresponds to g_j and if the minimum local score for g_j corresponds to f_i. Otherwise, the pairing is inconsistent and Exposé eliminates them from the set of `MAY`-pairs.

After eliminating inconsistent `MAY`-pairs, we normally end up with a bipartite matching (since the minimum scores in rows and columns are usually unique). If not, we can run a standard bipartite matching algorithm to generate the final pairs.

3) Grouping Related Matching Functions: We prefer matches in which both callers and callees are matched to isolated matches. Based on this intuition, given a set M of final matched pairs (which includes both `IS`-pairs and the consistent `MAY`-pairs), Exposé groups the library functions as follows. Given a matched pair (f_1, g_2) in M, if there is another pair (f_2, g_2) in M such that f_1 calls f_2 and g_1 calls g_2, Exposé puts f_1 and f_2 in the same group. This is achieved programmatically by finding undirected cycles of length four with edges formed by function calls or pairings. Figure 3 shows a trivial example in which two groups exist, with the first group comprising f_1 and f_2 and the second group comprising f_3.

To give preference to matched groups over isolated matches, Exposé scales the cosine distance of each matched function f_i by dividing by the size of its group (recall that lower scales indicate higher similarity).

To compute an overall composite matching score between a library and an application, Exposé uses a simple method for ranking the matches: the final distance score is obtained by taking the average of the scaled cosine distances of the pairs in the final matched set, M. This simple method worked quite well for the purpose of ranking, as discussed next.

V. RESULTS AND EVALUATION

The experimental objectives are to understand how well Exposé performs qualitatively (in terms of ranking matching applications to a given library) and quantitatively (in terms of timing performance). Our experiments were conducted on machines with Intel Xeon processor at 2.50 GHz running Redhat Enterprise Linux release 5.4.

TABLE IV: 10 smallest distance scores using `libpng` as test library. Smaller scores indicate better matches between the library and the applications.

File	Score	Elapsed (min)	Size (kb)
feh-static	0.273	0.235	482.10
feh-shared	0.288	0.181	435.50
sash	0.290	1.599	595.04
bash	0.306	3.754	663.69
md5sum	0.312	0.210	18.93
sln	0.314	0.885	449.10
gawk-3.1.5	0.330	1.188	293.05
tar	0.342	0.776	188.20
sed	0.353	0.739	93.18
cpio	0.355	0.821	51.53

A. Quality of Ranking of Applications

For this experiment, we wanted to evaluate the quality of Exposé's rankings. One challenge we encountered was the lack of ground truth, since it would not have been possible for us to manually analyze thousands of binaries. To address the challenge, we conducted the experiment in two parts. In the first part (controlled), Exposé generated rankings for a given library against a small set of applications that we knew to be using the library. In the second part (uncontrolled), Exposé computed rankings for another library against a large set of applications for which we had little information about their library usages. We used different libraries for the two parts to study Exposé's ability to work for different libraries.

1) Controlled: We used `libpng v1.2.43` as the test library. The library is commonly used by applications for processing PNG images. The version of the library we used contained a buffer overflow vulnerability CVE-2010-1205 resulting from insufficient checks on buffer space in the function `png_push_process_row`. For the test applications, we compiled two versions of a simple image viewer application called `feh`. The first version, `feh-static`, was statically linked with the test library, while the second version, `feh-shared`, used dynamic linking and was used as a control. In addition, we included 128 executable binaries from the `/bin` directory of one of the RedHat machines. These applications were unlikely to use `libpng`. We ran Exposé for finding code re-use of the `libpng` test library in the set of test applications. The results for the 10 smallest scores are shown in Table IV. From the results, we observe that `feh-static` correctly had the nearest distance score.

2) Uncontrolled: To test Exposé's ability to correctly rank a large set of applications, we used 2,927 unique application binaries (determined by their SHA-1 hash) from DVL v1.5 [44], a Linux distribution based on Slackware containing a large collection of applications for practicing purpose by penetration testers. We used `zlib v1.2.3` as the test library. The choice of the library was intentional because it contained identifiable signatures. Our techniques did not make use of the signature; the signature was only used to estimate the ground truth.

To verify the correctness of the results, we used `Clamscan` to analyze the same set of applications for the known signature. `Clamscan` is an open-source tool for scanning applications for known signatures, and it is more commonly used as part of `ClamAV`, an anti-virus engine [45]. The library contained 88 functions. Using `Clamscan`, we identified 10 application binaries that made use of `zlib`, all using version 1.2.3: `depmod`, `modinfo`, `modprobe`, `rateup`, `sash`, `rsync`,

TABLE V: 15 smallest distance scores using `zlib` as test library.

File	Score	Elapsed (min)	Size (kb)
dpkg-deb	0.07	1.352	174.40
rsync	0.09	5.762	262.93
sash	0.10	2.159	595.04
insmod.static.old	0.11	2.827	692.80
modinfo.old	0.14	1.283	107.64
depmod.old	0.14	1.317	131.54
depmod	0.14	1.256	88.28
modinfo	0.14	1.175	64.12
modprobe	0.14	1.258	77.17
rateup	0.14	1.416	337.43
insmod.old	0.14	1.369	174.46
ddd	0.19	6.755	2707.53
psp	0.21	0.044	22.38
snort	0.22	2.164	676.97
jpegtran	0.22	0.379	77.69

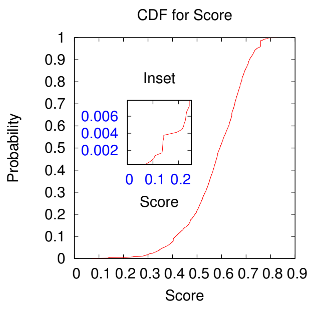

Fig. 4: Cumulative distribution of distance scores. The inset shows the probabilities for scores between 0 and 0.25.

`depmod.old`, `insmod.old`, `insmod.static.old`, and `modinfo.old`.

Table V ranks 15 application binaries with the nearest distance scores. We observe that the 10 application binaries detected by `Clamscan` were amongst the top 11 scores. There was one surprise. `dpkg-deb`, which was not detected by `Clamscan`, was ranked first.

We used `Clamscan` on `dpkg-deb` using the signature for `zlib v1.2.2` and a positive result was returned. Additionally, we examined dumping out the strings in the binary and found that it contained the string "inflate 1.2.2 Copyright 1995-2004 Mark Adler", indicating that it was possibly linked with `zlib v1.2.2` at some point. This was further confirmed by disassembling `dpkg-deb` with IDA Pro [46] and comparing the instructions. The small similarity score (indicating a good match) for version 1.2.3 was achieved because the set of `zlib` functions called by `dpkg-deb` was a subset of the functions that did not vary significantly between versions 1.2.2 and 1.2.3.

Figure 4 shows the cumulative distribution of the distance scores for all 2,927 binary applications. The distance scores for the true positives were ranked above that for all other applications except `dpkg-deb`. This indicates that Exposé can be helpful in significantly narrowing down the number of binaries that need to be examined for a closer match.

B. Library Versions and Compiler Options

Next, we did experiments to study Exposé's ability to distinguish among library versions and its sensitivity to common compiler optimization options.

TABLE VI: Distance scores of applications compiled with different `zlib` versions compared with `zlib v1.2.3`.

App	`zlib` version	Score
app1	1.1.3	0.034
app2	1.1.4	0.034
app3	1.2.1	0.010
app4	1.2.2	0.010
app5	1.2.3	0.006
app6	1.2.4	0.020

TABLE VII: Distance scores between the 11 applications with smallest distance scores and different `zlib` versions. The smallest distance scores for each application are **bolded**. The correct version used by the application is marked with an asterisk (*).

Application	v1.1.4	v1.2.2	v1.2.3	v1.2.4
dpkg-deb	0.300	0.073*	**0.070**	0.285
rsync	0.270	0.092	**0.089***	0.303
sash	0.267	0.154	**0.103***	0.295
insmod.static.old	0.443	0.156	**0.108***	0.683
modinfo.old	0.441	0.205	**0.135***	0.426
depmod.old	0.407	0.209	**0.137***	0.431
depmod	0.431	0.214	**0.137***	0.441
modinfo	0.443	0.209	**0.139***	0.441
modprobe	0.436	0.211	**0.139***	0.448
rateup	0.432	0.211	**0.142***	0.454
insmod.old	0.443	0.220	**0.142***	0.431

1) Distinguishing Library Versions: Our goal here was to determine if Exposé could be effective in identifying specific library version use without resorting to specialized methods such as computing differences among libraries or generating signatures for the differences. As an initial test case, we compiled a test application statically linked with different versions of `zlib` and ran Exposé against `zlib v1.2.3`. The functions called were the same and included modified functions across the different library versions. The results are shown in Table VI. We found that the distance score increases as the version is further away from the current version, indicating that Exposé can be effective in distinguishing variations in the binaries.

We next evaluated Exposé with different versions of `zlib` on the 11 applications (including dpkg-deb) that evidently made use of `zlib`. The results are shown in Table VII. We observe that the 10 applications that were statically linked with `zlib v1.2.3` have the smallest distance scores with `zlib v1.2.3`. However, `dpkg-deb` also has its smallest distance score with `zlib v1.2.3`. In addition, its distance score with `zlib v1.2.2` is also relatively small. We found four functions in both `v1.2.2` and `v1.2.3` to have `IS`-pairs: `compressBound`, `deflatePrime`, `deflateBound` and `zcalloc`. These matches contributed the most to the small distance score. All these functions were found to be unchanged between the two versions.

To further analyze the relative contributions of `IS`-pairs and `MAY`-pairs to the final scores, Table VIII shows the number of `IS`-pairs, number of `MAY`-pairs and the final score that would have resulted if only `MAY`-pair scores were used for the 11 applications for the various library versions. Our findings indicate that the count of `MAY`-pairs or the score from `MAY`-pairs, while useful as a component for generating a ranking of applications, is less effective in distinguishing among versions, probably because n-gram matching is not likely to produce significant differences for versions of the same library. On the

TABLE VIII: The table shows the tuples (number of IS-pairs, number of MAY-pairs, average MAY-pair scores) for the top 11 binaries in the same order as Table V.

File	v1.1.4	v1.2.2	v1.2.3	v1.2.4
dpkg-deb	0, 38, 0.34	4, 39, 0.35	4, 34, 0.33	1, 36, 0.36
rsync	1, 27, 0.30	3, 33, 0.33	3, 32, 0.34	1, 42, 0.35
sash	0, 31, 0.30	2, 39, 0.36	3, 42, 0.36	1, 46, 0.33
insmod.static.old	0, 34, 0.30	2, 41, 0.36	3, 46, 0.37	0, 21, 0.68
modinfo.old	0, 26, 0.48	2, 27, 0.47	3, 33, 0.47	1, 31, 0.47
depmod.old	0, 24, 0.44	2, 29, 0.48	3, 31, 0.47	1, 32, 0.47
depmod	0, 26, 0.46	2, 30, 0.47	3, 31, 0.47	1, 33, 0.47
modinfo	0, 23, 0.46	2, 26, 0.47	3, 29, 0.47	1, 28, 0.47
modprobe	0, 25, 0.45	2, 26, 0.47	3, 30, 0.47	1, 30, 0.48
rateup	0, 30, 0.48	2, 35, 0.47	3, 41, 0.47	1, 45, 0.48
insmod.old	0, 32, 0.47	2, 34, 0.49	3, 37, 0.48	1, 36, 0.47

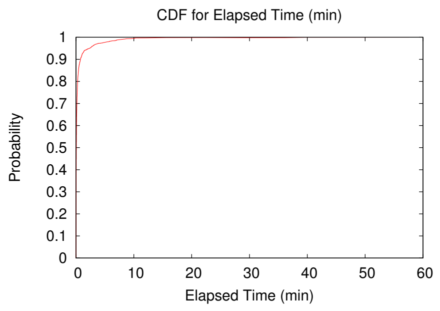

Fig. 5: Cumulative distribution of elapsed times.

other hand, the count of IS-pairs, even if small, is a very good indicator for identifying the likely library version.

We note that when we attempted to use semantic matching based on symbolic execution to the entire set of library functions, the matching task took too long (it did not finish). Our results suggest that restricting the semantic-match approach to a promising set of candidate functions, based on an appropriate criteria (e.g., same number of inputs/out-degrees or low cyclomatic complexity), can still produce good match results, including identifying the right library version, while significantly enhancing scalability.

2) Robustness under Varying Compiler Options: To study the robustness of Exposé under different compiler options, we compiled the test application using different compiler options. We verified that the compiled applications were different using `bsdiff`, a tool for generating patch files between binaries. We obtained the scores 0.031, 0.008, 0.006, and 0.054 for compiler options O1, O2, O3, and Os respectively. The distance score is the smallest when the application was compiled with the same compiler option as `zlib`, i.e., O3. Comparing with the results in Table V, the distances obtained for applications using other compiler options are also relatively small.

C. Timing Performance

Figure 5 shows the cumulative distribution of the elapsed times. The elapsed times do not include the times taken for disassembling the binaries since that will vary depending on the disassembler's performance. We used IDA Pro [46] as our disassembler. Exposé analyzed 97.68% and 99.48% of the binaries within five and 10 minutes respectively, with the longest analysis taking 57.62 minutes. Our results demonstrate the scalability of Exposé clearly.

VI. Conclusion

Identifying code re-use has many important security applications including detection of illegitimate software usage and vulnerable or buggy code re-use. Previous efforts towards solving the problem largely focused on detecting code re-use in source code. With Exposé, we aim to provide a practical solution for detecting potential binary code re-use, which is challenged by the lack of symbols, function inlining, and compiler-induced instruction variations.

Exposé determines candidate function pairs from a library and an application for semantic matching based on the number of input parameters, out-degree, function size, and cyclomatic complexity. We also use function level n-gram analysis to determine matching for pairs that are not amenable to symbolic execution. Thus, for each function in the library, we have either one of the three results: a semantic match, a syntactic match with a distance measure, or no match to a function in an application. When applied to a large number of applications for the given library, these results are summarized into a score for ranking the match quality between the library and the application.

To evaluate the ranking quality, we used Exposé to identify an application statically linked with `libpng` and placed with a set of 128 applications not known to be statically linked with `libpng`. In this controlled experiment designed to overcome the lack of ground truth, Exposé ranked the application statically linked with `libpng` at the top. Using another test library, `zlib`, Exposé ranked 2,927 applications, with the top 10 out of 11 applications also found to use `zlib` by a signature scanner. Upon manual analysis, the top ranked application that was not detected by the signature scanner was found to be linked with an earlier version of the library that contained similar functions as the test library. When we varied the compiler options, and used different versions of a test library, Exposé generated the shortest distances (or very close to that value) between applications and the library variants they were using. Exposé analyzed 97.68% and 99.48% of the binaries within five and 10 minutes respectively.

Looking forward, challenges remain for detecting malign binary code re-use. For example, with the growing popularity of mobile computing devices, such as tablets, detecting malign binary code re-use across different platforms may uncover common security issues.

References

[1] A. Monden, S. Okahara, Y. Manabe, and K. Matsumoto, "Guilty or Not Guilty: Using Clone Metrics to Determine Open Source Licensing Violations," *IEEE Software*, pp. 42–47, 2011.

[2] B. H. Ng, X. Hu, and A. Prakash, "A Study on Latent Vulnerabilities," in *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010), New Delhi, Punjab, India, October 31 - November 3, 2010*. IEEE, 2010, pp. 333–337.

[3] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire os distributions," in *IEEE Symposium on Security and Privacy*, 2012, pp. 48–62.

[4] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.30

[5] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, pp. 654–670, 2002.

[6] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Transactions on Software Engineering*, vol. 32, pp. 176–192, 2006.

[7] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *ICICS '08: Proceedings of the 10th International Conference on Information and Communications Security*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 238–255.

[8] J. Jang, D. Brumley, and S. Venkataraman, “Bitshred: feature hashing malware for scalable triage and semantic analysis,” in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS ’11. New York, NY, USA: ACM, 2011, pp. 309–320. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046742

[9] X. Hu, T. cker Chiueh, and K. G. Shin, “Large-scale malware indexing using function-call graphs.” in *ACM Conference on Computer and Communications Security*, E. Al-Shaer, S. Jha, and A. D. Keromytis, Eds. ACM, 2009, pp. 611–620. [Online]. Available: http://dblp.uni-trier.de/db/conf/ccs/ccs2009.html#HuCS09

[10] T. W. Reps, J. Lim, A. V. Thakur, G. Balakrishnan, and A. Lal, “There’s plenty of room at the bottom: Analyzing and verifying machine code,” in *CAV*, 2010, pp. 41–56.

[11] S. Wolchok, R. Yao, and J. A. Halderman, “Analysis of the Green Dam Censorware System,” University of Michigan, Tech. Rep. CSE-TR-551-09, 2009.

[12] M. Madou, B. Anckaert, and K. De Bosschere, “Code (De) Obfuscation,” 2005.

[13] M. Madou, L. Van Put, and K. De Bosschere, “Loco: an interactive code (de)obfuscation tool,” in *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, ser. PEPM ’06. New York, NY, USA: ACM, 2006, pp. 140–144. [Online]. Available: http://doi.acm.org/10.1145/1111542.1111566

[14] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, 1st ed. Addison-Wesley Professional, 2009.

[15] H.-i. Lim, H. Park, S. Choi, and T. Han, “Detecting theft of java applications via a static birthmark based on weighted stack patterns,” *IEICE - Trans. Inf. Syst.*, vol. E91-D, pp. 2323–2332, September 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1522931.1522933

[16] H. il Lim, H. Park, S. Choi, and T. Han, “A method for detecting the theft of Java programs through analysis of the control flow information,” *Inf. Softw. Technol.*, vol. 51, pp. 1338–1350, September 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558382.1558414

[17] H. Tamada, M. Nakamura, A. Monden, and K.-I. Matsumoto, “Java birthmarks - detecting the software theft,” *IEICE - Trans. Inf. Syst.*, vol. E88-D, pp. 2148–2158, September 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1184637.1184747

[18] H. Tamada, M. Nakamura, and A. Monden, “Design and evaluation of birthmarks for detecting theft of java programs,” in *In Proc. IASTED International Conference on Software Engineering*, 2004, pp. 569–575.

[19] D. Schuler, V. Dallmeier, and C. Lindig, “A dynamic birthmark for java,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE ’07. New York, NY, USA: ACM, 2007, pp. 274–283. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321672

[20] G. Myles and C. Collberg, “K-gram based software birthmarks,” in *Proceedings of the 2005 ACM symposium on Applied computing*, ser. SAC ’05. New York, NY, USA: ACM, 2005, pp. 314–318. [Online]. Available: http://doi.acm.org/10.1145/1066677.1066753

[21] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, “N-gram-based detection of new malicious code,” in *Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02*, ser. COMPSAC ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 41–42. [Online]. Available: http://portal.acm.org/citation.cfm?id=1025118.1025582

[22] J. D. Igor Santos, Yoseba K. Penya and P. G. Bringas, “N-Grams-based FIle Signatures for Malware Detection,” 2009.

[23] J. Jang and D. Brumley, “BitShred: Fast, Scalable Code Reuse Detection in Binary Code,” Carnegie Mellon University, Tech. Rep., 2009.

[24] K. L. Verco and M. J. Wise, “Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems,” in *Proc. of 1st Australian Conference on Computer Science Education*. ACM, 1996, pp. 86–95.

[25] S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing: Local algorithms for document fingerprinting.” in *SIGMOD Conference*, A. Y. Halevy, Z. G. Ives, and A. Doan, Eds. ACM, 2003, pp. 76–85. [Online]. Available: http://dblp.uni-trier.de/db/conf/sigmod/sigmod2003.html#SchleimerWA03

[26] H. W. Kuhn, “The hungarian method for the assignment problem,” *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955.

[27] S. Choi, H. Park, H.-i. Lim, and T. Han, “A static birthmark of binary executables based on api call structure,” in *Advances in Computer Science ASIAN 2007. Computer and Network Security*, ser. Lecture Notes in Computer Science, I. Cervesato, Ed. Springer Berlin / Heidelberg, 2007, vol. 4846, pp. 2–16, 10.1007/978-3-540-76929-3_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-76929-3_2

[28] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. ichi Matsumoto, “Dynamic software birthmarks to detect the theft of windows applications,” in *International Symposium on Future Software Technology (ISFST 2004)*, 2004.

[29] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, “Behavior based software theft detection,” in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS ’09. New York, NY, USA: ACM, 2009, pp. 280–290. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653696

[30] A. Moser, C. Kruegel, and E. Kirda, “Limits of static analysis for malware detection,” *Computer Security Applications Conference, Annual*, vol. 0, pp. 421–430, 2007.

[31] S. Treadwell and M. Zhou, “A heuristic approach for detection of obfuscated malware,” in *Proceedings of the 2009 IEEE international conference on Intelligence and security informatics*, ser. ISI’09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 291–299. [Online]. Available: http://portal.acm.org/citation.cfm?id=1706428.1706495

[32] K. Alzarouni, D. Clark, and L. Tratt, “Semantic malware detection,” Department of Computer Science, King’s College London, Tech. Rep. TR-10-03, Feb. 2010.

[33] M. D. Preda, M. Christodorescu, S. Jha, and S. K. Debray, “A semantics-based approach to malware detection,” *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 5, 2008.

[34] S. Bucur, “Scalable Automated Testing Using Symbolic Execution,” 2009.

[35] T. Hansen, P. Schachte, and H. Sndergaard, “State Joining and Splitting for the Symbolic Execution of Binaries,” in *Runtime Verification*, ser. Lecture Notes in Computer Science, S. Bensalem and D. Peled, Eds. Springer Berlin / Heidelberg, 2009, vol. 5779, pp. 76–92. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04694-0_6

[36] X. Xiao, X. song Zhang, and X. da Li, “New approach to path explosion problem of symbolic execution,” in *Pervasive Computing Signal Processing and Applications (PCSPA), 2010 First International Conference on*, sept. 2010, pp. 301 –304.

[37] R.-G. Xu, “Symbolic Execution Algorithms for Test Generation,” Ph.D. dissertation, University of California, Los Angeles, 2009.

[38] T. J. McCabe, “A complexity measure,” in *Proceedings of the 2nd international conference on Software engineering*, ser. ICSE ’76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 407–. [Online]. Available: http://portal.acm.org/citation.cfm?id=800253.807712

[39] J. C. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, pp. 385–394, July 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[40] V. Ganesh and D. L. Dill, “A decision procedure for bit-vectors and arrays,” in *Computer Aided Verification (CAV ’07)*. Berlin, Germany: Springer-Verlag, July 2007.

[41] D. Brumley and I. Jager, *The BAP Handbook*, May 2009.

[42] G. M. Penn. (2007) Calling Conventions Part I (Passing Integral Arguments). Online. [Online]. Available: http://w00zl3.net/files/calling_conventions.pdf

[43] M. Nagao and S. Mori, “A new method of n-gram statistics for large number of n and automatic extraction of words and phrases from large text data of japanese,” in *In COLING-94*, 1994, pp. 611–615.

[44] C. C. UG. Damn vulnerable linux. [Online]. Available: http://www.damnvulnerablelinux.org/

[45] (2011) ClamAV. Sourcefire Inc. [Online]. Available: http://www.clamav.net/lang/en/

[46] Hex-Rays. The IDA Pro Disassembler and Debugger. Online. Hex-Rays SA. Boulevard de la Sauvenire, 30 4000 Lige Belgium. [Online]. Available: http://www.hex-rays.com/idapro/

Appendix

Procedure 2 Compute local score between functions f_i and g_j.

```
function CalcLocalScore ($f_i, g_j, d$)
if $d$ >DEPTHLIMIT then
    return $c_{i,j}$
end if
$c_p$ = CalcParentLocalScore($f_i, g_j, d$)
$c_c$ = CalcChildLocalScore($f_i, g_j, d$)
$l_{i,j} = (c_{i,j} + c_p + c_c)/3$
return $l_{i,j}$
end function

function CalcParentLocalScore ($f_i, g_j, d$)
for all parent $f_p$ of $f_i$ do
    for all parent $g_p$ of $g_j$ do
        $m[f_p][g_p]$ = CalcLocalScore( $f_p, g_p, d+1$ )
    end for
    $m[f_i][g_p]$ = CalcLocalScore ($f_i, g_p, d+1$)
end for
$l_{i,j}$ = CalcMunkres( $m[f_i][g_p]$ )
return $l_{i,j}$
end function

function CalcChildLocalScore ($f_i, g_j, d$)
for all child $g_c$ of $g_j$ do
    for all child $f_c$ of $f_i$ do
        $m[f_c][g_c]$ = CalcLocalScore( $f_c, g_c, d+1$ )
    end for
    $m[f_c][g_j]$ = CalcLocalScore ($f_c, g_j, d+1$)
end for
$l_{i,j}$ = CalcMunkres( $m[f_i][g_p]$ )
return $l_{i,j}$
end function
```