Labeling Library Functions in Stripped Binaries

Emily R. Jacobson, Nathan Rosenblum, Barton P. Miller
Computer Sciences Department
University of Wisconsin
Madison, WI 53706 USA
{jacobson,nater,bart}@cs.wisc.edu

ABSTRACT

Binary code presents unique analysis challenges, particularly when debugging information has been stripped from the executable. Among the valuable information lost in stripping are the identities of standard library functions linked into the executable; knowing the identities of such functions can help to optimize automated analysis and is instrumental in understanding program behavior. Library fingerprinting attempts to restore the names of library functions in stripped binaries, using signatures extracted from reference libraries. Existing methods are brittle in the face of variations in the toolchain that produced the reference libraries and do not generalize well to new library versions. We introduce *semantic descriptors*, high-level representations of library functions that avoid the brittleness of existing approaches. We have extended a tool, UNSTRIP, to apply this technique to fingerprint wrapper functions in the GNU C library. UNSTRIP discovers functions in a stripped binary and outputs a new binary, with meaningful names added to the symbol table. Other tools can leverage these symbols to perform further analysis. We demonstrate that our semantic descriptors generalize well and substantially outperform existing library fingerprinting techniques.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, and reengineering*

General Terms

Algorithms, Experimentation, Measurement

Keywords

Reverse engineering, Static analysis, Stripped binaries

PASTE'11, September 5, 2011, Szeged, Hungary.

1. INTRODUCTION

For most computer users, program binaries are a black box, essential to running the program but not interesting as artifacts in and of themselves. For code analysts, however, binary code provides a wealth of information about the program. Binary code analysis provides details about the content of the program's code (instructions, basic blocks, functions, and modules), structure (control and data flow), and data (global and stack variables), all of which can provide insight into the program's runtime behavior and intent. Such information is essential for binary modification and translation, performance profiling and modeling, debugging, computer security, and forensics—domains in which source code representations may not be appropriate (such as measuring the performance impact of compiler optimizations) or available (malware authors rarely distribute source code).

While the information contained in the binary is crucial for these tasks, analyzing and understanding program binaries is more difficult than analyzing the original source code. This difficulty stems in large part from the binary's lack of types, expressive syntax, comments, and formatting present in the source code. This lack, combined with a modern compiler's complex transformation of the code [1], adds significant cost and to the analysis of a binary. The problem is compounded by the difficulty of simply finding code in program binaries; binaries are often stripped of their debugging symbols, have functions that share code and are non-contiguous in memory, and have non-code (such as tables, strings, and padding) interspersed with code [6, 20, 22].

A major part of a program's behavior is its interaction with the operating system. Knowing how the program interacts with the operating system can significantly increase the analyst's ability to understand the semantics, intent, and performance of the program, and is of particular importance in malware analysis [2, 4]. Most programs do not directly invoke the *system calls* that define the operating system interface. Instead, they call *wrapper functions* provided by standard system libraries; such wrapper functions invoke the required system call or calls, often through the use of trap instructions. In this paper, we present a robust *library fingerprinting* technique that identifies wrapper functions based on their interaction with the system call interface. Library fingerprinting restores names to library code that has been statically linked into program binaries by identifying recognizable characteristics—fingerprints—of standard library functions. A direct benefit of identifying wrapper functions is that we help to understand and add meaning to complex functions that call these routines.

Existing library fingerprinting techniques use simple pattern matching to identify functions [13, 23]. For example, the widely used IDA Pro disassembler stores patterns similar to byte-level regular expressions that it has derived from existing libraries. These patterns can be used only to find library code that is nearly bytewise identical to the library from which the patterns are derived; such approaches are brittle in the face of code produced by different compiler versions or build options, and do not generalize well across library versions. Our goal is to generate patterns that are tolerant of these naturally occurring binary differences. There is an essential tension, however, between specificity and generalizability: signatures must capture the details that differentiate functions, but must abstract away minute differences between versions.

Unlike the byte-level patterns of compiled code, a library function's semantics are not affected by different compiler and build options. As expressions of a public interface, library function semantics also tend to be relatively stable over time, as changes can break compatibility with existing code; this observation applies particularly to wrapper functions, whose semantics are determined by the system call interface. The system calls invoked by wrapper functions provide a basis for behavioral library fingerprinting that generalizes across binary code variations and the evolution of system standard libraries.

System calls do not fully determine the semantics of wrapper functions, however, and alone are insufficient as fingerprints. For example, the wrapper functions `setcontext` and `swapcontext` in the GNU C Library both invoke the *sigprocmask* system call on Linux platforms and are indistinguishable under this criterion. This multiplicity of invocations arises from the fact that system calls often implement broad functionality, which is refined to a particular task by a wrapper function — usually by invoking the system call with specific, fixed arguments. These parameters further define the semantics of wrapper functions.

However, extracting semantic patterns from compiled libraries is complicated by variations introduced by the compiler, particularly *function inlining*. Depending on the build options and the compiler version, a wrapper function that is composed of several auxilliary methods at the source level may be realized in the binary either as several distinct functions or as a single unit. Patterns derived from inlined code will not apply to non-inlined libraries, and vice versa.

In this paper, we introduce *semantic descriptors* that capture the high-level semantics of wrapper functions using the characteristics of system call invocations. Our approach yields two contributions. First, we use semantic descriptors to create fingerprints for wrapper functions; these fingerprints, based on the essential, invariant characteristics of system calls, can generalize across different library versions and compiler variations. Second, we define a flexible pattern matching algorithm that identifies specific wrapper functions based on library fingerprints. We use our technique to form fingerprints for the GNU C Library (`glibc`) and to restore wrapper function names to stripped program binaries, which informs our high-level workflow:

1. We obtain binaries for several versions of the `glibc` library as a reference set; these binaries are obtained from several Linux distributions, and are compiled using several compiler toolchains. The code variations among these libraries allow us to determine the generalizability of semantic descriptor-based fingerprints.

2. Using the ParseAPI parsing library [18], we extract an instruction representation and control flow graph (CFG) for each *exported* function in a particular `glibc` binary. Exported functions define the public interface of the GNU C Library, where we anticipate behavioral stability across library versions. During parsing, we identify wrapper functions by recording direct invocations of system calls.

3. For each wrapper function, we construct a semantic descriptor based on the invariant characteristics of the invoked system call(s), and record the function fingerprint. Invariant characteristics include the system call name and any concrete parameter values.

4. To identify wrapper functions in program binaries, we search for functions that directly invoke system calls and construct a semantic descriptor; we then compare the descriptor against a database of library fingerprints. The fingerprint matching algorithm is a two-stage process that tolerates slight variations in fingerprints.

We have implemented semantic descriptor-based library fingerprinting as an extension to UNSTRIP, a tool that discovers functions in stripped binaries and adds generic names to the symbol table. Our extensions allow UNSTRIP to add meaningful names to GNU C library wrapper functions that are discovered in stripped binaries. The benefit of UNSTRIP is that it creates a new binary with a symbol table, so any tool that depends on a symbol table will be able to leverage this information.

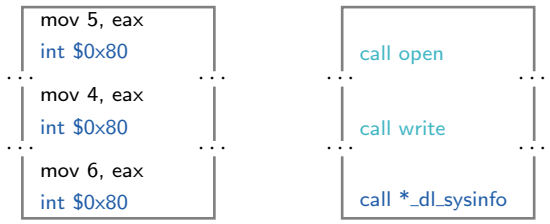
We compared semantic descriptors against the byte-level fingerprinting approach of the industry standard IDA Pro Disassembler, testing whether fingerprints generated from a particular version of `glibc` can precisely identify wrapper functions in a different version, or in a version produced by a different compiler toolchain. Our results show that our technique generalizes well, achieving high accuracy (99%) and offering a significant improvement (167%) over IDA Pro.

In the following sections, we provide an overview of library fingerprinting challenges (2), and describe semantic descriptors (3) and the fingerprint matching process (4). We evaluate our fingerprinting techniques on a large set of `glibc` binaries (5), and conclude with a review of the related literature (6).

2. PROBLEM OVERVIEW

Library fingerprinting is a pattern recognition problem that is complicated by the binary code from which fingerprints are generated. Binary code reflects not only the program functionality defined by the source code, but also the varying properties of the compiler toolchain and build environment, such as optimization strategies and conditional compilation [1]. These variations introduce a plethora of binary code artifacts that complicate our process. In particular, our approach addresses three complications: function inlining, code reordering, and minor code variations.

Function inlining is illustrated in Figure 1. Although the source code for `sethostid` is identical in `glibc` 2.3.2 and `glibc` 2.5, the binary code for each reveals differences. In



(a) in `glibc` 2.3.2 (b) in `glibc` 2.5

Figure 1: An example of function inlining in the wrapper function `sethostid`.

`glibc` 2.3.2, invocations of the system calls *open* and *write* are inlined, while in `glibc` 2.5, `sethostid` instead calls the `open` and `write` wrapper functions. Function inlining complicates fingerprinting because patterns must be insensitive to these compositional changes. To address this, our fingerprinting technique first generates the semantic descriptor for a wrapper function, and then recursively generates semantic descriptors for any wrappers called by the function. The fingerprint for the wrapper function is then the union of its semantic descriptor with those generated recursively. Thus, the fingerprints for the two versions of `sethostid` are identical, though the binary code representations differ.

Code reordering occurs because, at the binary level, both the layout of instructions and the order in which they are encountered during parsing can vary. These variations do not impact function behavior. For instance, conditional branch reordering, in which the branch instruction is inverted and the taken and not taken branches are reversed, alters the binary code but does not affect behavior. Semantic descriptors should not take these orderings into account. To accomplish this, we store system calls in a semantic descriptor in unordered sets.

Minor code variations are caused by both the build environment and source code alterations as the library evolves. Conditional compilation options or added utility functionality do not affect the intrinsic behavior of the function, but they do impact the binary code representation. For example, `accept` in `glibc` 2.2.4 only invokes a single system call, *socketcall*. However, in `glibc` 2.11.1, `accept` exhibits the same behavior with slightly different source code: here, a multithreaded option has been added, resulting in an additional system call, *futex*, and a second instance of *socketcall*. To address these code variations, we use a flexible pattern matching technique, described in Section 4, to evaluate possible identifications. This allows our technique to correctly locate matches within our database, even when the semantic descriptors may be slightly different.

3. SEMANTIC DESCRIPTORS

We construct semantic descriptors representing wrapper functions from binary code. Semantic descriptors are defined by the set of system calls invoked by a function and their arguments. More precisely, we define a *system call invocation* to be the tuple $\sigma = \langle s_i, a_1, \ldots, a_{K(i)} \rangle$, where s_i is a particular system call and $a_{1:K(i)}$ are the arguments of that system call. A semantic descriptor is a *multiset*[1] of M system call invocations $\{\sigma_1, \ldots, \sigma_M\}$. The *descriptor database* (DDB) represents the set of descriptors D and wrapper functions W; we define a *fingerprint* function $f : D \mapsto W$ that maps from a descriptor $d \in D$ to a particular wrapper function w for function identification.

Figure 2 illustrates the semantic descriptor generation process. We use the ParseAPI parsing library to locate system calls, which appear in three flavors in 32-bit Linux (see Figure 3). We then identify wrapper functions, those that contain one or more system calls. For each wrapper function w, we build its semantic descriptor based from the system call invocations in w. Once we have processed the system calls in w, we recursively generate the semantic descriptors for any wrapper functions called by w. This algorithm is presented in Figure 4.

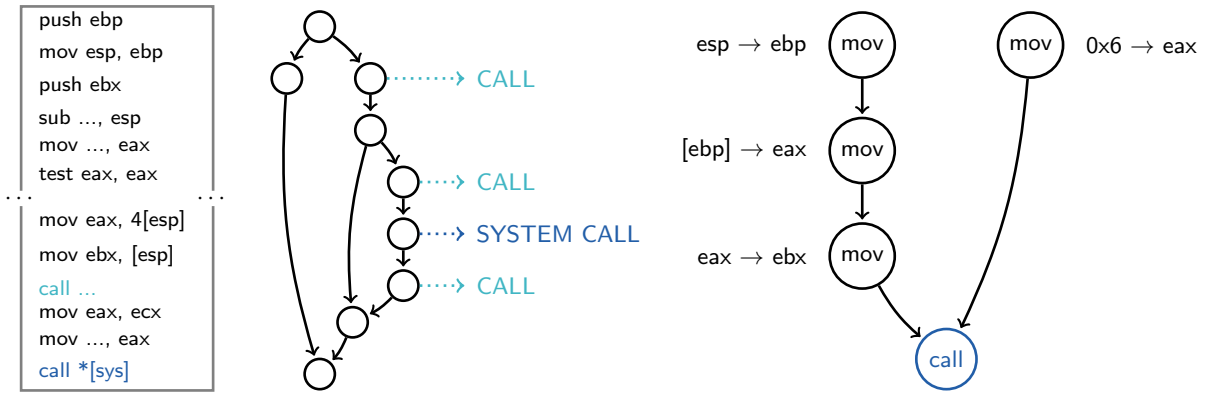
We extract system call names and arguments using the DataflowAPI library [17]. This library provides backward slicing [5, 14] and symbolic evaluation [3, 7] facilities that we use to determine the system call number. First, a backward slice is performed from the system call invocation site; this slice is based on `%eax`, the hardware register in which the system call number is stored. Once the slice is generated, we perform symbolic evaluation using the ROSE [8] instruction semantics specifications and extract the value stored in `%eax`. Based on the system call number, we determine how many arguments (and what types) should be passed to the function using the Linux kernel interface specifications. We use a second backward slice to identify any concrete argument values. For system calls with six arguments of fewer, argument values are placed in the hardware registers `%ebx`,`%ecx`, `%edx`, `%esi`, `%edi`, and `%ebp`; this process is encapsulated by the function GetArgs in Figure 4. We simply omit any arguments for which a concrete value cannot be statically determined. For each system call, a tuple is generated and added to w's semantic descriptor.

The recursive descriptor generation process is important for two reasons. First, it allows semantic descriptors to be robust in the face of function inlining. Second, it allows our technique to incorporate information about these called functions in a symmetrical manner. The semantic details of each called function provide important distinguishing information. For instance, both `getloadavg` and `gethostid` call the wrapper function `open`, but recursively generating `open`'s semantic descriptor in each case reveals distinguishing information: in `getloadavg`, `open` is called with `/proc/loadavg` as its first parameter, the pathname, whereas in `gethostid`, `/etc/hostid` is used.

We assemble fingerprints from a set of reference libraries to form the DDB. These libraries span several versions of `glibc`, have been constructed with different build environments, and come from different Linux distributions. Although the semantic descriptors capture invariant characteristics of functions, the use of a set of reference libraries allows us to be robust to naturally occurring differences, such as minor code variations.

We use the learning mode of UNSTRIP to create these fingerprints. For each wrapper function, w, in the library, UNSTRIP generates the semantic descriptor, d, and stores the fingerprint, $f(d) = w$. For each reference library, we store the set of wrapper function fingerprints. These sets are then combined to form the DDB. Because the semantic descriptors record invariant function characteristics, most fingerprints will be identical across many library versions; the

[1]A multiset extends the definition of a set by allowing multiple, identical elements.



(a) Assembly (b) CFG (c) Slice

Figure 2: The semantic descriptor generation process. From the raw instructions of the function (a), we generate the control flow graph (b) and identify all system calls and wrapper function calls. We slice (c) backwards over the instructions in the CFG to determine the system call number and any arguments that can be concretely identified using symbolic evaluation. The system calls and concrete parameter values forms the semantic descriptor for this function.

DBB stores a single copy of these duplicated fingerprints. Further, the DDB does not store library information; our objective is to identify wrapper functions, not the library version from which the fingerprint came.

4. PATTERN MATCHING

We identify wrapper functions in a binary using flexible pattern matching. Our approach balances the two goals of precisely labeling wrapper functions and providing as much information as possible about the contents of the binary; this is a tradeoff between leaving functions unidentified and providing multiple potential labels. We flexibly identify functions by (1) using a relaxed pattern matching criterion in which inexact matches are allowed and (2) returning all matching fingerprints when multiple matches are found. Providing multiple potential labels is preferred to an unidentified wrapper function.

We generate semantic descriptors for wrapper functions in the binary. Then, for each wrapper function, we search the DDB for a match in two stages. First, we search for an exact match. If none exist, we move to the second stage, where we heuristically apply a *coverage* criterion. Coverage examines how much of a semantic descriptor can be explained by a potential match; we use this heuristic to locate the most similar match in the database. We define coverage as

$$coverage(A, B) = \frac{|A \Cap B|}{|B|},$$

where fingerprint A is a potential match for semantic descriptor B, and $A \Cap B = \{b \in B \mid b \in A\}$. Coverage is non-symmetric. For example, the `accept` wrapper function has the following semantic descriptors in two different instances of `glibc`:

(d_1) $\{\langle$socketcall,5$\rangle\}$
(d_2) $\{\langle$socketcall,5$\rangle, \langle$socketcall,5$\rangle, \langle$futex$\rangle\}$.

The coverage depends on which descriptor is used as the fingerprint: $coverage(d_1, d_2) = 2/3$, while $coverage(d_2, d_1) = 1$. This asymmetry allows us to more accurately evaluate how each potential match relates to the semantic descriptor and find the unique, correct match more frequently than simple intersection.

We compute the fingerprint that maximizes coverage of the descriptor for the candidate wrapper function. If two or more different fingerprints are tied under the coverage criterion, we select the fingerprint with the smallest descriptor; this choice expresses a preference for simpler explanations. As for exact fingerprint matches, more than one fingerprint may match a given function candidate. We return the set of

(a)
```
mov ..., eax
int $0x80
```

(b)
```
mov ..., eax
call *%gs:SYSINFO_OFFSET
```

(c)
```
mov ..., eax
call *_dl_sysinfo
```

Figure 3: Three forms of trap-based system calls in 32-bit Linux: (a) a special interrupt instruction, (b) an indirect call through the gs register, and (c), an indirect call through the `_dl_sysinfo` symbol.

function SEMANTICDESCRIPTOR(w)
  $d \leftarrow \emptyset$
  for all $s \in$ SYSCALLS(w) **do**
    $a_1, \ldots, a_{k(s)} \leftarrow$ GETARGS(s, w)
    $d \leftarrow d \cup \{\langle s, a_1, \ldots, a_{k(s)}\rangle\}$
  for all $c \in$ CALLS(w) **do**
    if SYSCALLS(c) $\neq \emptyset$ **then**
      $d \leftarrow d \cup$ SEMANTICDESCRIPTOR(c)

Figure 4: An algorithm for building semantic descriptors.

matching fingerprints in this case. To prevent false matches, we require the final match or matches to have coverage of at least .5.

Infrequently, multiple matches exist. In these cases, when two or more fingerprints have identical semantic descriptors or coverage results for a given wrapper function, we label the wrapper function with the set of possible names discovered by our pattern matching algorithm. In our experience, however, indistinguishable fingerprints are usually closely related, and thus assigning multiple names still provides useful information about the wrapper function. For example, the wrapper functions `setcontext` and `swapcontext` faciliate the manipulation of program state for signal handling. Both invoke a single system call, *sigprocmask*, with one identical argument. Labeling a wrapper function with both possible identities still provides the analyst with useful information for most tasks.

5. EVALUATION

We have incorporated semantic descriptor generation and flexible pattern matching as an extension to UNSTRIP [19]. There are now two operation modes: learning and identification. Learning mode takes as input a reference library and generates a DDB based on fingerprints from this library. Identification mode labels wrapper functions in a stripped binary. For each wrapper function in this binary, UNSTRIP generates a semantic descriptor and applies our pattern matching technique to obtain a match or set of matches. If no such match exists, the function remains unidentified. UNSTRIP outputs a new binary with this function identification information added to the symbol table.

We conducted a multidimensional evaluation of the generalizability of semantic descriptor-based fingerprints for `glibc` using UNSTRIP. We compared the results with the Fast Library Identification and Recognition Technology (FLIRT) from the industry standard IDA Pro Disassembler (version 6.0) [12]. Our results demonstrate that our technique is able to precisely identify wrapper functions. Further, our semantic descriptors have significantly better generalizability characteristics than the current state of the art.

5.1 Methodology

The binary code in a particular library can differ due to changes to the source code, compilation toolchain, or build environment. To evaluate how well different library fingerprinting techniques handle such variation, we constructed three `glibc` data sets, each representing a different source of variation:

Toolchain Different compilers and compiler versions can have pronounced effects on binary code [21]. We compiled `glibc` 2.5 source code with GNU C Compiler (GCC) versions 3.4.4., 4.0.2, 4.1.2, and 4.2.1 to construct libraries with varying toolchain characteristics.

Library version Source code changes that accrete as the library evolves can significantly impact the resulting binary, both due to the code changes and also due to constraints on the compiler that can be used to build a particular library version. We use `glibc` versions 2.2.4, 2.3.2, 2.3.4, 2.5, and 2.11.1 from the Red Hat Entriprise Linux (RHEL) distribution and different corresponding GCC versions to construct these libraries.

Distribution Variations introduced by library vendors derive from vendor-specific code patches and the particular build environment, including the compiler version. We collected pre-compiled versions of glibc 2.11.1/2 from the Fedora, Mandriva, OpenSuse, and Ubuntu Linux distributions.

We conducted a three-dimensional study, comparing the library fingerprinting performance of UNSTRIP and IDA Pro on each. For each data set, we:

1. constructed a set of `glibc` libraries within this dimension,
2. computed the *ground truth* for this dimension, the set of wrapper functions common to all libraries within the dimension,
3. compiled a test binary for each `glibc` instance by statically linking in all functions from the library,
4. used the learning mode of UNSTRIP to extract fingerprints from a single library instance and build a DDB,
5. applied the identification mode of UNSTRIP to the binaries linked against the remaining library instances within the data set, and
6. evaluated how well UNSTRIP was able to identify wrapper functions in these binaries. We use the ground truth within the dimension as the basis for evaluation.

Because we are searching a database for information about particular functions, we evaluate function identification as an information retrieval task. We measure *true positives* (correctly identified wrapper functions), *false positives* (incorrectly identified wrapper functions), and *false negatives* (unidentified wrapper functions). We then use *precision* and *recall* metrics to evaluate the effectiveness of an algorithm. The precision

$$precision = \frac{true\ positives}{true\ positives + false\ positives}$$

measures how well the fingerprinting technique avoids assigning spurious labels, while recall

$$recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

measures how well a technique finds all relevant information.

5.2 Results

We present evaluation results for each dimensional study, with precision and recall results for both UNSTRIP and IDA Pro. Because UNSTRIP provides multiple labels for a function when we cannot distinguish between potential matches, we present results both with and without these multiple matches. When evaluating results in which multiple matches are allowed, we count a wrapper function as correctly identified if the correct wrapper function label is present.

Table 1 shows the results from the toolchain study. UNSTRIP achieves high precision (.98) and recall (1.00) when multiple matches are included; when we exclude these multiple-match results, UNSTRIP still achieves recall of .90. In contrast, IDA Pro achieves high precision, but recall between .61 and .79. These results demonstrate that while our approach is able to generalize across variations, IDA Pro is not.

Reference Library	UNSTRIP (mult)		UNSTRIP		IDA Pro	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
3.4.4	0.98	1.00	0.98	0.90	1.00	0.61
4.0.2	0.99	1.00	0.99	0.90	1.00	0.63
4.1.2	0.98	1.00	0.98	0.90	1.00	0.79
4.2.1	0.98	1.00	0.98	0.90	1.00	0.66

Table 1: Results for the *toolchain* study. Reference libraries are identified by GCC version, and results are averaged across the remaining libraries in the set. For UNSTRIP, multiple match results are presented first, followed by results without multiple matches.

Byte-level patterns almost always preclude the possibility of false positives, so IDA Pro's precision is high. However, such an approach cannot overcome the binary differences introduced by compiler version.

Table 2 shows the results of the library version study. UNSTRIP achieves similar results to the toolchain study; when we exclude multiple-match results, recall is slightly higher than before (.93-.96), because UNSTRIP is able to locate a single best match for more wrapper functions. IDA Pro, however, has lower recall than the previous study, between .12 and .31. These results demonstrate that our approach is able to generalize across binary differences caused by changing library versions. In the evolution of `glibc`, we observe that our approach is able to generalize both forwards and backwards along this continuum.

Table 3 shows the results from the distribution study. As with the first two dimensional studies, UNSTRIP achieves high precision and recall when multiple matches are included, and slightly lower recall (.84-.87) when multiple matches are excluded; IDA Pro achieves low recall, between .20 and .34. We observe more dramatic code changes between libraries from different distributions; both source code and also build environment differences have been introduced by the distributor. Our approach is able to generalize across these changes.

Byte-level fingerprinting techniques degrade rapidly as compiler or source code changes are introduced. Figure 5 depicts the accuracy of using fingerprints from `glibc` 2.2.4 to predict the other more recent libraries within the library version study. In contrast to IDA Pro, UNSTRIP maintains a high level of accuracy across all versions, even as the distance between library versions increases.

In each study, we allowed UNSTRIP to return multiple matches; in almost all cases, the correct label was found, always within the top 3 results. These results provide practical benefits to an analyst. Further, we recorded the frequency and size of multiple-label sets returned by UNSTRIP; overall, 8% of the identifications included indistinguishable functions, with an average set size of 2.05. Often, when multiple matches exist, they still provide reasonable information to the analyst. `pwrite` and `pwrite64` are an example of such a set; these two functions have nearly identical semantics but accommodate different parameter sizes (here, `pwrite64` takes a 64-bit file offset). `setegid` and `setresgid` are another: both manipulate the group ID of the current process. These types of patterns are common. Infrequently, the set of multiple matches is more ambiguous: for instance, `close` is identified as `close`, `netlink_close`, and `alloc_dir`. Even in this case, however, the analyst is provided with a small set of matches, among which is the correct label.

Reference Library	UNSTRIP (mult)		UNSTRIP		IDA Pro	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
2.2.4	1.00	0.99	1.00	0.96	1.00	0.12
2.3.2	1.00	0.99	1.00	0.94	1.00	0.31
2.3.4	1.00	0.99	1.00	0.94	1.00	0.31
2.5	0.99	0.99	0.99	0.93	1.00	0.17
2.11.1	0.99	0.99	0.99	0.93	1.00	0.17

Table 2: Results for the *library version* study. Reference libraries are identified by `glibc` version.

Reference Library	UNSTRIP (mult)		UNSTRIP		IDA Pro	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
Fedora	0.99	0.99	0.99	0.84	1.00	0.22
Mandrivia	0.99	1.00	0.99	0.87	1.00	0.34
OpenSuse	0.99	1.00	0.99	0.87	1.00	0.20
Ubuntu	0.99	1.00	0.99	0.87	1.00	0.34

Table 3: Results for the *distribution* study. Reference libraries are identified by vendor.

These results demonstrate that our approach is able to generalize across library versions in several dimensions. We achieve both high precision and recall, illustrating that we are able to correctly identify almost all wrapper functions, neither introducing false labels or leaving functions unidentified. We presented results both with and without multiple matches. Because we focus on supplying the analyst with as much function information as possible, UNSTRIP always returns multiple matches if they exist; it is up to the analyst to decide what to do with this information. IDA Pro's byte-level approach is unable to identify most wrapper functions, across all three dimensions of variation. These results support our hypothesis that a behavioral pattern approach is necessary to provide generalizability.

5.3 Discussion

Our evaluation focuses on binaries containing only library functions; however, we expect these results will generalize to more common binaries, containing a mixture of application- and library-level functions, also. Although many application-level functions invoke system calls, they do so via the standard interface provided by wrapper functions. Because we only fingerprint functions that directly invoke system calls, i.e., wrapper functions, our technique will not spuriously label application-level functions.

Currently, there are two limitations to our approach. First, our fingerprinting approach is limited to wrapper functions. However, this technique could be expanded to apply to other library functions, using wrapper functions as a foundation and leveraging additional structural information [4, 11, 15]. Second, our study was performed on Linux and this technique has not been evaluated on other platforms; however, wrapper functions exist on other platforms as well, so our technique could easily be extended.

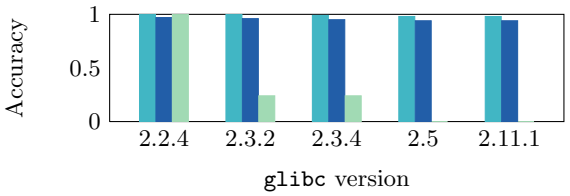


Figure 5: Identification results for UNSTRIP with multiple matches (■), UNSTRIP without multiple matches (■), and IDA Pro (■) using fingerprints from `glibc` 2.2.4 and the binaries from the library version study.

UNSTRIP incurs a few additional limitations because of the libraries on which it is built. UNSTRIP assumes the ability to extract information from a binary, requiring that ParseAPI locate functions in the binary code. Function location identification is a well-studied problem and can be difficult. IDA Pro faces this challenge as well. Further, UNSTRIP relies on the ROSE instruction semantics specifications for symbolic evaluation, which are currently only available for 32-bit binaries.

6. RELATED WORK

Previous library fingerprinting approaches have relied on byte-level pattern matching, forming signatures from the bytes underlying the initial instructions of library functions [12, 23]. The IDA Pro disassembler builds signatures of library functions [12] from the first 32 bytes of a function, with wildcards for bytes that vary when the library is loaded; Van Emmerik [23] proposed a similar approach using variable-length signatures. Byte-level fingerprints can be made arbitrarily precise by extending the signatures; as we have shown in this paper, however, such approaches generalize poorly to minor variations across different binary instances of a library.

Alternative binary code representations that elide byte-level details have been used in polymorphic malware detection and to detect changes between versions of program binaries [9, 10, 16]. These techniques use a graph-based representation of code, abstracting away much of the instruction-level detail in favor of *structural* properties of program. This representation is effective for detecting the introduction or removal of code due to patches [10], but is less well suited for describing system call wrapper functions, whose semantics are determined by instruction-level properties like parameter values; a control flow-based representation also remains subject to minor code variations between library versions.

Several approaches have used semantic or behavioral patterns to characterize binary code in the anti-malware community [4, 11, 15]. These approaches identify patterns in the externally visible behavior of programs, such as interactions with the operating system (through system calls or standard libraries) or manipulation of the filesystem; for example, Fredrikson et al. [11] form malware specifications based on the sequence of system calls and their arguments observed at runtime. While there is some overlap between the objective of malware identification and library fingerprinting (recognizing specific functionality while tolerating binary code variation), the techniques are largely orthogonal. Our library fingerprinting technique statically identifies system call wrapper functions in program binaries; behavioral malware identification is an essentially dynamic task, identifying patterns in the ordered execution trace of malicious code.

7. CONCLUSION

We have presented a technique for accurately identifying library wrapper functions using semantic descriptors and flexible pattern matching. In contrast to traditional byte-pattern techniques, we take a semantic approach, using invoked system calls and their associated arguments to create fingerprints. Our tool, UNSTRIP, allows analysts to both extract fingerprints from binaries and also label wrapper functions in stripped binaries. The results of the evaluation of UNSTRIP strongly support our hypothesis that a behavioral approach to library fingerprinting is necessary to provide generalizability. Our approach achieves on average 99% accuracy when identifying wrapper functions in a variety of `glibc` instances. Any binary tool that relies on symbol table information can benefit from UNSTRIP.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and suggestions. This research funded in part by Department of Homeland Security grant FA8750-10-2-0030 (funded through AFRL), National Science Foundation grants CNS-0716460 OCI-1032341, and Department of Energy grants DE-SC0004061, DE-SC0003922, and DE-SC0002154.

References

[1] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *Verified Software: Theories, Tools, Experiments*. Springer-Verlag, 2007.

[2] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2009.

[3] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Trans. Softw. Eng.*, 5(4):402–417, 1979.

[4] M. Christodorescu, S. Jha, and C. Krugel. Mining specifications of malicious behavior. In *Proceedings of the Sixth Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 5–14, Dubrovnik, Croatia, 2007.

[5] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Proc. International Conference on Software Maintenance*, pages 188–195, Bari, Italy, October 1997.

[6] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software–Practice and Experience*, 25(7), 1995.

[7] P. Coward. Symbolic execution systems-a review. *Software Engineering Journal*, 3(6):229–239, Nov 1988.

[8] D. J. Quinlan et al. ROSE Compiler Project. `http://www.rosecompiler.org`.

[9] T. Dullien and R. Rolles. Graph-based comparison of executable objects. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, June 2005.

[10] H. Flake. Structural comparison of executable objects. In *Conference Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2004)*, Dortmund, Germany, July 2004.

[11] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, Berkeley, California, May 2010.

[12] I. Guilfanova and DataRescue. Fast library identificatiion and recognition technology. `http://www.hex-rays.com/idapro/flirt.htm`, 1997.

[13] Hex-Rays. IDA Pro disassembler. `http://www.hex-rays.com/idapro`.

[14] A. Kiss, J. Jasz, G. Lehotai, and T. Gyimothy. Interprocedural static slicing of binary executables. In *Source Code Analysis and Manipulation*, Amsterdam, The Netherlands, September 2003.

[15] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zho, and X. Wang. Effective and efficient malware detection at the end host. In *Eighteenth USENIX Security Symposium*, Montreal, Canada, August 2009.

[16] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Eighth International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, Seattle,WA, September 2005.

[17] Paradyn Project. Dyninst 7.0. 2011. URL `http://www.paradyn.org/html/dyninst7.0-features.html`.

[18] Paradyn Project. ParseAPI: An application program interface for binary parsing. 2011. URL `http://paradyn.org/html/parse0.9-features.html`.

[19] Paradyn Project. UNSTRIP. 2011. URL `http://paradyn.org/html/tools/unstrip.html`.

[20] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt. Learning to analyze binary computer code. In *23rd conference on Artificial Intellegence (AAAI '08)*, Chicago, IL, July 2008.

[21] N. E. Rosenblum, B. P. Miller, and X. Zhu. Extracting compiler provenance from program binaries. In *9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '10)*, Toronto, Ontario, Canada, June 2010.

[22] H. Theiling. Ecxtracting safe and precise control flow from binaries. In *7th Conference on Real-Time Computing Systems and Applications (RTCSA '00)*, Washington, DC, December 2000.

[23] M. Van Emmerik. Signatures for library functions in executable files. Technical Report 2194, Queensland University of Technology, 1994.