Binary Diffing via Library Signatures

Andrei Rimsa
CEFET-MG
Belo Horizonte, Brazil
andrei@cefetmg.br

Anderson Faustino da Silva
UEM
Maringá, Brazil
afsilva@uem.br

Camilo Santana
UFMG
Belo Horizonte, Brazil
camilo.santana@dcc.ufmg.br

Fernando Magno Quintão Pereira
UFMG
Belo Horizonte, Brazil
fernando@dcc.ufmg.br

***Abstract*—Binary diffing is the problem of determining whether two binary programs originate from the same source code. Binary diffing tools are used to identify malware, plagiarism, or code theft. Many instances of binary diffing assume an adversarial setting, where a malicious actor transforms binary code by changing the compiler. Traditional diffing techniques rely on statistical similarity analysis, often leveraging stochastic models. However, recent studies have shown that these classifiers perform poorly in the face of adversarial code transformations. To mitigate this scenario, this paper introduces a new diffing technique that is resilient against current obfuscation approaches. We propose comparing executables by matching their library signatures (libsig). A program's library signature is the sequence of calls it makes to functions outside its `.text` section. The proposed classifier, LIBSIG, is faster than off-the-shelf alternatives, such as LTRACE, and, like it, works on stripped binaries or binaries running with address space layout randomization (ASLR) enabled. Furthermore, in contrast to LTRACE, LIBSIG can be engineered to detect even library calls that bypass conventional application binary interface patterns. Our experiments on the GNU Core Utilities demonstrate that LIBSIG remains robust against obfuscators like KHAOS and OLLVM, as well as typical optimization patterns, outperforming binary diffing approaches such as SAFE, BINDIFF, or ASM2VEC.**

***Index Terms*—Library, Signature, Invocation**

I. INTRODUCTION

Binary diffing is a technique used to analyze differences between two binary executables. This is a problem with many practical applications, some of which include, for instance, (i) Patch Analysis: comparing patched and unpatched binaries to understand what vulnerabilities were fixed and whether they can be exploited [57, 60]; (ii) Malware Analysis: detecting the presence of potentially modified versions of malware code [2, 36]; (iii) Software Similarity Analysis: identifying code reuse between different binaries to detect software plagiarism or licensing violations [19, 58]; (iv) Firmware Analysis: analyzing updates in firmware to detect tampering or backdoors [37, 59].

Many techniques have been used in the implementation of binary diffing tools. However, in recent years, there has been a noticeable shift in approach. Traditional binary diffing methods were primarily deterministic, relying on well-defined heuristics and mathematical properties, such as graph isomorphism and subgraph matching [4], and syntax- or hash-based comparisons [42]. In contrast, recent approaches, such as SAFE [34], ASM2VEC [11], GEMINI [56], and VEXIR2VEC [51], leverage machine learning to improve the scalability of older techniques.

Diversification Strikes Back: Despite the diverse techniques used to build binary diffing tools, recent studies have shown that state-of-the-art approaches can be evaded with relative ease. In 2021, Ren et al. [43] demonstrated that simply altering optimization sequences could reduce the precision of seven leading binary classifiers, including ASM2VEC [11] and BINSLAYER [4], to less than 45%. Later, Zhang et al. [67] showed that basic source-code-level obfuscation techniques could substantially degrade the accuracy of neural network-based binary diffing tools, such as ASTNN [64] and TBCCD [61]. In response, Damásio et al. [10] demonstrated that these simple source-code modifications were ineffective against classifiers operating at the binary level. However, the same study, corroborated by da Silva et al. [9], found that OLLVM [25], an obfuscator built on the LLVM intermediate representation, remained effective at evading machine learning-based classification. In 2023/24, Zhang et al. [65, 66] introduced KHAOS, an even more powerful LLVM-based obfuscator capable of inter-procedural code transformations. Finally, in 2025, VenkataKeerthy et al. [51] showed that diversification techniques, such as changing the compiler, compiler version, or optimization level, could be even more successful than obfuscation in reducing the precision of tools like SAFE, BINDIFF [1], and DEEPBINDIFF [12].

The Key Idea of this Paper: This paper presents a dynamic analysis technique designed to withstand the diversification approaches discussed earlier. Said method matches binaries based on their *Library Signature*, which, as explained in Section III-A, represents the sequence of library calls made during a program's execution. Section III-B details how to extract this signature, even from stripped binaries and in the presence of protections such as *address space layout randomization (ASLR)* [50]. To extract a program's library signature, we introduce LIBSIG, which we have implemented in two ways: either on top of VALGRIND [38], or using the RTLD-AUDIT API. To illustrate its effectiveness, Section IV introduces a program classification game in which the goal is to match two groups of programs compiled differently. Using this benchmark, Section V demonstrates that LIBSIG remains robust against commercial and academic obfuscation techniques currently in use. LIBSIG's design embodies two contributions:

- **Idea:** LIBSIG uses the sequence of library calls made by an executable as its signature. This approach does not require the instrumentation of the library code; thus, in contrast to previous dynamic analysis methods [21, 63, 68],

979-8-3315-9288-2/26 © 2026 IEEE

LIBSIG benefits from the presence of library calls[1].

- **Engineering:** This paper demonstrates how to distinguish between library code, the so-called invisible instructions, and visible code [69]. To this end, we have implemented two versions of LIBSIG, one on top of VALGRIND, and another on top of RTLD-AUDIT. The former handles a larger family of executable files and works on more operating systems; the latter is faster.

Summary of Results: LIBSIG performs binary matching by comparing library signatures extracted from the execution of two binaries. Section V evaluates the performance of the two versions of LIBSIG on 102 programs from the GNU Core Utilities (`CoreUtils`). As shown in Section V-B, both versions outperforms binary diffing tools such as SAFE [34], ROUXINOL [16], ASM2VEC [11], and BINDIFF [13, 17]. Section V-C shows that LIBSIG is practical: even though it requires running programs, at least when tested in `CoreUtils`, it operates faster than the static classifiers. Section V-D shows that LIBSIG is also faster than other VALGRIND plugins. Furthermore, it is almost twice as fast as LTRACE, a standard Linux utility that could be used to track library calls. Additionally, it tracks calls involving eager linking (§III-A1) or runtime linking (§III-A3), while LTRACE does not.

II. LIBRARY SIGNATURE

Programs often invoke external code during their execution. External code include *Standard Libraries*, such as `libc` for functions like `atoi` and `printf`; third-party libraries, like OPENSSL or ZLIB; and runtime support libraries tied to specific languages or toolchains, such as `libstdc++`. Example 1 shows a program that invokes external functions.

Example 1. *The C program in Figure 1 calls three external functions:* `atoi` *and* `printf` *from the C Library (*`libc`*), and* `sqrt` *from the Math Library (*`libm`*).*

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <math.h>
04  void main(int argc, char* argv[]) {
05    int n = atoi(argv[1]);
06    double s = sqrt(n);
07    printf("square root of %d is %g\n", n, s);
08  }
```

Fig. 1. Program in C that calls three library functions.

To avoid imprecisions, in this paper, we call "*User Code*" the instructions in the `.text` section of an executable binary. This section contains code written by the programmer – which comes from the program's "*source code*". But it also contains: (i) statically linked library code; (ii) startup/runtime code (e.g., `_start`, `__libc_start_main`); and (iii) possibly compiler-generated code (e.g., C++ virtual tables, unwind handlers, etc). Given these observations, we define "*External Code*" and the "*Library Signature*" of a program as follows:

[1]Previous work often claims that "*If library code is not instrumented, [then] the simulation is only partial and far less trustworthy.*" [68]

Definition 1 (Library Signature)**.** *The* External code *are instructions located outside the* `.text` *section of the executable binary. The* Library Signature *of an executable P and its input I, is the sequence of calls to external code observed during execution.*

Example 2. *Figure 2 shows the library signature of an execution of the program in Figure 1. This signature contains three functions:* `atoi`*,* `sqrt`*, and* `printf`*, which lay outside the executable's* `.text` *section. For comparison, Figure 2 shows the address of* `main`*, which lays inside that section.*

```
      .text segment starts at: 0x563fe8647000
      .text segment ends at:   0x563fe8648295

 user code  Address of main:   0x563fe8648140
external code Address of atoi:   0x7ff9709f35b0 ┐
external code Address of sqrt:   0x7ff970bb1780 │ Library Signature
external code Address of printf: 0x7ff970a10c90 ┘
```

Fig. 2. Library signature of the program in Figure 1, when executed with input `"0"`.

The library signature of a program's execution may serve as a behavioral *fingerprint*, allowing a program to be distinguished from other binaries. This signature tends to remain stable across different compilers, and even in the presence of common obfuscation techniques such as control-flow flattening or instruction substitution. Owing to this robustness, library signatures can be adapted as a means to solve binary identification tasks, as illustrated in Section II-A.

A. A Use-Case Example: License Violation

Many open-source licenses, such as the GNU General Public License (GPL) [18], impose strong copyleft requirements. These requirements include the obligation to disclose source code when redistributing binaries derived from GPL-licensed software. Violations of these terms can lead to both community enforcement and litigation, as Example 3 shows.

Example 3. *In Versata Software, Inc. v. Ameriprise Financial, Inc. [52], the court considered behavioral and structural similarities between software products as circumstantial evidence of license infringement. Similar arguments were accepted in Jacobsen v. Katzer [24], which established that non-compliance with open-source license terms may constitute a copyright violation.*

Detecting license violations in practice is challenging, particularly when infringing parties attempt to disguise their binaries. A common evasion strategy involves recompiling GPL-licensed code using aggressive or nonstandard compiler optimization sequences, such as combinations that do not correspond to typical flags like `-O1`, `-O2`, or `-O3` [43].

Binary Diffing as Sequence Matching: As demonstrated in Section V, our approach leverages library signatures to detect similarities under such adversarial conditions. The method proceeds in two steps. First, we execute the target binary with a representative input set I and record the resulting sequence of

external calls, yielding a library signature $\mathsf{libsig}_P(I)$. Second, we compare this signature against those produced by candidate open-source programs using the same inputs. To quantify similarity, we treat each library signature as a string over the alphabet of observed library functions and compute the Edit Distance between them. While some optimizations or substitutions may alter a few calls (e.g.: `atoi` $\rightarrow$ `strtol`), a typical execution may involve hundreds or thousands of external calls. In practice, the majority of calls remain consistent across compilation variants. A low edit distance between two signatures, $\mathsf{libsig}_P(I)$ and $\mathsf{libsig}_Q(I)$, provides strong evidence that P and Q were compiled from closely related source code. Although such behavioral similarity is not conclusive legal proof, it serves as an effective screening tool to flag potential license violations and guide more detailed structural or symbolic analyses.

III. Extracting the Library Signature

The symbols within a program's code, such as variable and function names, must be resolved to actual memory addresses for the program to execute. This resolution process is called *linking*, and it can occur either during compilation (static linking) or at runtime (dynamic linking), as explained in Definition 2.

Definition 2 (Linking Time)**.** Static linking *is the process of resolving external symbols at compile time. The linker copies the necessary code (e.g., library functions) into the final binary's* `.text` *section, making the resulting executable self-contained and independent of external libraries.* Dynamic linking*, in contrast, defers the resolution of external symbols to runtime. The executable includes references to shared libraries (e.g.,* `.so` *files on Linux,* `.dylib` *files on macOS, and* `.dll` *files on Windows), which are loaded into memory and then linked, where its symbols such as function names are associated to their corresponding code, just before or during program execution.*

Combining Definitions 1 and 2, the library signature of a program is the sequence of calls to dynamically linked functions made during its execution. When *static linking* is used, external code is merged onto the `.text` section of the program, and thus there will be no library signature. We discuss limitations of this approach in Section V-E. Section III-A explains how different *dynamic linking* strategies operate, and Section III-B describes how we can extract a program's library signature, regardless of the strategy it employs.

A. Dynamic Linking Strategies

A program may use different strategies to dynamically bind calls to external functions to their actual implementations. This section explains the three most common strategies, using, to this end, the program in Figure 1. Examples shall assume that the program was compiled with `clang` for an x86-64 processor running Linux. However, similar strategies can be found in different combinations of compilers, processors and operating systems.

1) Eager Linking: In eager linking, the resolution of external function symbols is performed before the program's `main` function begins execution. When the dynamic linker (e.g., `ld.so` on Linux) loads the binary into memory, it eagerly resolves all external references by locating their corresponding code addresses in shared libraries and updating the relevant entries in the *Global Offset Table* (GOT)[2]. As a result, function calls in the binary point directly to their final destinations from the beginning of execution. This strategy does not benefit from the use of the *Procedure Linkage Table* (PLT)[3], which may postpone the resolution of symbols as required by the program during execution. Thus, each function invocation incurs less overhead since they were resolved beforehand. Eager linking can be enabled in compilers like `gcc` or `clang` using the `-fno-plt` flag.

Example 4. *Figure 3 illustrates how eager linking resolves the call to the* `sqrt` *function seen in Figure 1. The* `.got` *section holds the Global Offset Table (GOT), which contains the addresses to library functions, while the* `.text` *section contains the program's instructions, including the three indirect calls that reference entries in the GOT. Before execution, the GOT contains zero-filled placeholders. During program loading, i.e. before main is called, the* RTLD *(*RunTime LoaDer*) resolves these entries. It first locates the shared libraries required by the program (in this case,* `libc.so`*,* `libm.so`*, and possibly others) and loads them into memory. Next, it determines which portions of the libraries implement the functions referenced by the program. The function names are kept in the* `.rela.dyn` *section of the binary. Although this section is not shown in Figure 3, the names it holds can be inspected using tools such as* `objdump` *(with the* `-R` *option) or* `readelf` *(with the* `-r` *option). Figure 4 shows these relocation symbols as produced by* `objdump`*. These are called relocation records, since they need to be relocated for each execution of the program. Once all symbols are resolved, the execution proceeds as follows: the indirect call to* `sqrt` *loads its address from the GOT and transfers control to the corresponding function in memory (step 1); after the function completes execution it returns to the instruction immediately following that call (step 2).*

2) Lazy Linking: Lazy linking is the default strategy used by modern compilers, such as `clang` and `gcc`. This approach defers the resolution of external function symbols until the first time each function is called. At program startup, entries in the *Global Offset Table* point to stubs in the *Procedure Linkage Table*, which initially direct control to the dynamic linker. When external functions are invoked for the first time, the dynamic linker resolves their addresses and updates the corresponding GOT entry, replacing the stub's address.

[2]The *Global Offset Table* (GOT) is a section of a binary used in dynamically linked programs to store the runtime addresses of external symbols, allowing indirect access to shared library functions and global variables.

[3]The *Procedure Linkage Table* (PLT) is a section of a binary that facilitates the dynamic resolution of function calls by providing stubs that initially redirect control to the dynamic linker and are later updated to jump directly to the resolved function addresses stored in the GOT.

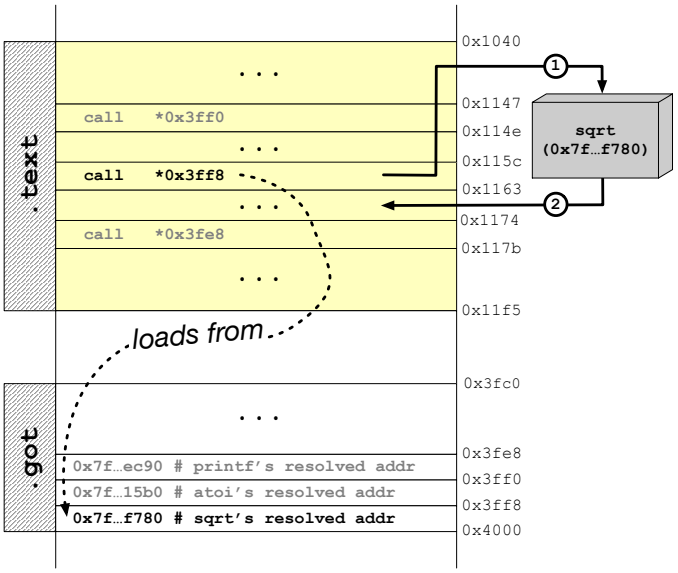

Fig. 3. Eager linking example for the invocation of the `sqrt` function.

```
OFFSET   TYPE                  VALUE
3fe8     R_X86_64_GLOB_DAT     printf@GLIBC_2.2.
3ff0     R_X86_64_GLOB_DAT     atoi@GLIBC_2.2.5
3ff8     R_X86_64_GLOB_DAT     sqrt@GLIBC_2.2.5
```

Fig. 4. Objdump snippet for the dynamic relocations records for the external functions used in Figure 1 when compiled without PLT.

Subsequent calls then bypass the PLT and jump directly to the resolved address. This approach reduces startup time by avoiding unnecessary symbol resolution for functions that may never be used during execution, albeit using more complex structures that require more auxiliary code. Example 5 clarifies some of these concepts.

Example 5. *Figure 5 illustrates how lazy linking tracks the resolution of the call to* `sqrt` *seen in Figure 1. The compiler initializes the* `.got.plt` *section of the binary with the addresses of handler stubs for each external function used by the program: in this example,* `atoi`*,* `sqrt`*, and* `printf`*. Note that the* `.got.plt` *entry for* `atoi` *already contains its resolved address, since it was invoked earlier. In contrast, the entries for* `sqrt` *and* `printf` *still point to their respective handler stubs, as these functions have not yet been called. When* `sqrt` *is invoked (step 1), control is transferred to the corresponding entry in the* `.plt` *section, which performs an indirect jump through the associated* `.got.plt` *entry. Because* `sqrt` *is being called for the first time, the jump leads to the address* `0x1056` *(step 2), which lies immediately after the jump instruction. At this point, the* `.plt` *entry pushes a function-specific index onto the stack; in this example,* `sqrt` *is assigned index* `2`*. Next, all first-time function calls in the* `.plt` *section jump to a shared lazy resolution handler located at address* `0x1020` *(step 3). This handler uses the index on the stack to lookup the corresponding*

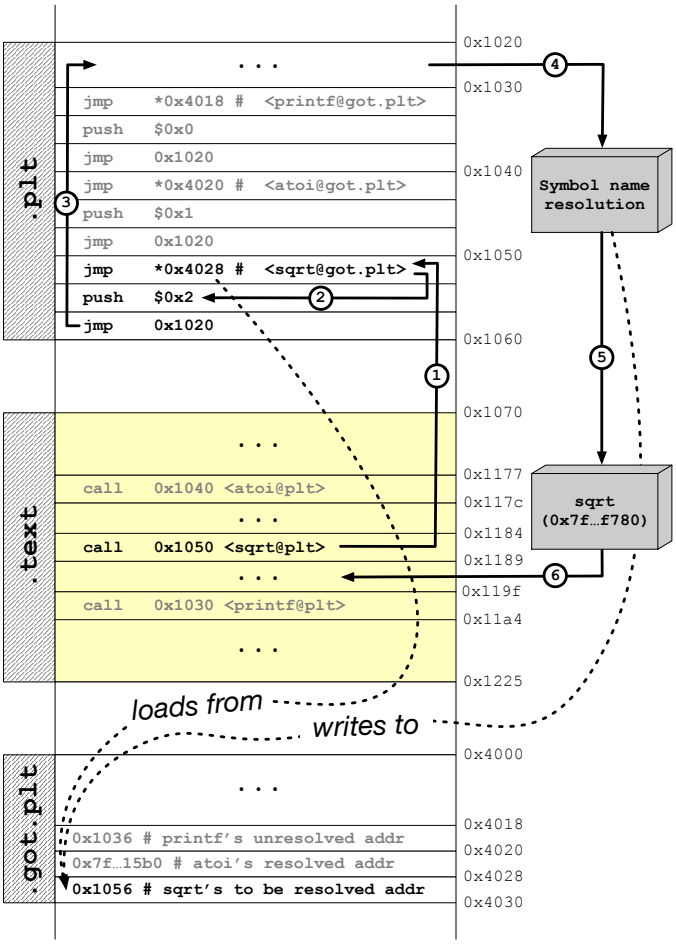

Fig. 5. Lazy linking example for the invocation of the sqrt function.

symbol name and resolve its address (step 4). Once the symbol has been resolved, the handler writes the resolved address of `sqrt` *into the appropriate* `.got.plt` *entry and transfers control to the target function (step 5). Eventually, when* `sqrt` *returns (step 6), control resumes at the instruction immediately following the original call, allowing the program to continue normal execution. Now, since the GOT holds the already resolved address, if this function is called a second time, it goes straight to its corresponding external code.*

3) Runtime Linking: A program may choose to employ another linking strategy: load and resolve these symbols at a later time in the execution. This is different from lazy linking: lazy linking is performed automatically by the interplay between the compiler and the dynamic linker, while runtime linking is program-controlled, as Example 6 shows.

Example 6. *Figure 6 shows a piece of C code that implements runtime linking using the* `dlopen` *and* `dlsym` *functions from the POSIX dynamic linking API. The program first requests that the library* `libm` *be loaded into memory by passing its file path to* `dlopen`*. The programmer can choose whether to resolve all symbols immediately at this point by passing the* `RTLD_NOW` *flag, or defer symbol resolution until the symbols*

are needed by passing the `RTLD_LAZY` flag. Once the library is loaded, the program requests the address of the function `sqrt` using `dlsym`, which searches for the symbol by its name. If `RTLD_NOW` was used, this symbol was already resolved during the `dlopen` call; otherwise, the resolution occurs at this point when `dlsym` is invoked. After the function address is retrieved, it can be called like any function pointer. When the library is no longer needed, it can be unloaded from memory using `dlclose`.

```
01  void* handle = dlopen("/lib/x86_64-linux-gnu/libm.so.6",
                          RTLD_NOW);
02  double (*sqrt)(double) = dlsym(handle, "sqrt");
03  double s = (*sqrt)(n);
04  dlclose(handle);
```

Fig. 6. This code snippet must replace the call to `sqrt` in Line 06 of Figure 1 to implement runtime linking.

B. Library Signature Extraction

Overview of LIBSIG: To track library calls, we have implemented LIBSIG, a tool that extracts the library signature of a program. LIBSIG is built on top of VALGRIND [38]; thus, it supports a wide range of architectures and operating systems. By using VALGRIND's dynamic instrumentation infrastructure, LIBSIG can observe transitions between code regions during program execution to record external library calls.

Tracking Code Regions: LIBSIG monitors the program's `.text` section (i.e., user code). Whenever execution crosses from this region outward, such as when calling external functions, LIBSIG logs the accessed address. This behavior corresponds to step 1 in Figures 3 and 5. By default, LIBSIG does not log when execution returns back into the observed region (step 2 in Figure 3 and step 6 in Figure 5). However, users can log these events by enabling them in LIBSIG if required. In other words, LIBSIG can be configured to record outward calls, inward returns, or both.

Address Resolution: To make the recorded addresses meaningful, LIBSIG resolves them into function names whenever possible. This step is necessary since binaries may load libraries at different addresses due to position-independent executables (PIE) and address space layout randomization (ASLR). LIBSIG leverages VALGRIND's native dynamic loader to resolve symbols from loaded libraries. However, in cases of lazy linking (§III-A2), some calls go through the `.plt` section, whose addresses are not resolved by the dynamic loader. In these situations, LIBSIG extracts relocation symbols from the binary via OBJDUMP to resolve function names, as shown in Figure 4. In this example, for lazy linking, the relevant relocation type is `R_X86_64_JUMP_SLOT` rather than `R_X86_64_GLOB_DAT`.

C. Multi-Threading

LIBSIG supports multi-threaded programs by leveraging VALGRIND's internal scheduler to track context switches. It records execution traces on a per-thread basis, ensuring that the library signature accurately captures the behavior of each thread independently. These per-thread sequences are then merged according to their observed execution order, as determined by VALGRIND's internal serialization model. Although VALGRIND does not assign explicit timestamps, it preserves the relative interleaving of events across threads. As a result, while correctness is preserved, VALGRIND's thread serialization can lead to significant performance overhead, particularly on multi-core systems where concurrent execution is not fully utilized.

This overhead could be mitigated by implementing LIBSIG atop a dynamic binary translation framework, such as INTEL PIN or DYNAMORIO. However, developing tools on these platforms is more complex, as one must explicitly manage thread interleavings, synchronization, and thread-local storage. To better illustrate these trade-offs, Table I compares the three frameworks in terms of their threading models, performance overhead, and suitability for our application. VALGRIND, which underpins LIBSIG, serializes all threads and executes them on a synthetic CPU. This design guarantees deterministic behavior, even in multi-threaded programs, but prohibits true parallel execution. Consequently, VALGRIND can impose substantial overhead on multi-core systems, especially for parallel workloads. By contrast, DYNAMORIO and INTEL PIN allow threads to execute concurrently on the host CPU using dynamic binary translation. This approach results in better scalability and lower overhead in multi-threaded scenarios. However, it does not guarantee that a parallel program will always produce the same library signature across executions, as the interleaving of calls may vary nondeterministically.

D. VALGRIND, RTLD-AUDIT *and* LTRACE

We have implemented two versions of LIBSIG: one based on VALGRIND, and another using RTLD-AUDIT. Additionally, Linux systems offer LTRACE, which provides functionality similar to the RTLD-AUDIT-based version of LIBSIG. RTLD-AUDIT exposes an auditing interface in the dynamic linker, allowing users to intercept symbol resolution events through a custom shared object loaded at program startup. LTRACE works by inserting breakpoints at entries in the Procedure Linkage Table (PLT), enabling it to intercept calls to dynamically linked functions at runtime. Both LTRACE and RTLD-AUDIT depend on the presence of a PLT; therefore, they are ineffective in scenarios involving *eager linking* (§III-A1) or *runtime linking* via `dlopen` (§III-A3). The VALGRIND-based version does not suffer from this shortcoming.

Table II compares the different approaches. In terms of **Precision**, the VALGRIND-based approach is more general, as it handles all the calling mechanisms discussed in Section III, whereas the other tools require the presence of a PLT in the executable. It is also the most portable across operating systems (**OS**), working on Linux and macOS (up to version 10.13 High Sierra); however, RTLD-AUDIT supports a wider range of computer architectures, as it can be used on any system featuring GLIBC's dynamic linker. The VALGRIND-based implementation also provides the highest level of **Control**, allowing us to restrict the analysis to specific regions of the

TABLE I
COMPARISON OF BINARY INSTRUMENTATION FRAMEWORKS WITH RESPECT TO MULTI-THREADING AND PERFORMANCE.

Tool	Thread Model	Performance Overhead	Suitability for LIBSIG
VALGRIND [38]	Serialized (one thread at a time)	High on multi-core systems	Simplifies instrumentation; portable but slower
DYNAMORIO [5]	Concurrent (native threading)	Medium to low	More efficient; requires concurrency handling
INTEL PIN [33]	Concurrent (native threading)	Medium	Similar to DynamoRIO; slightly more mature

TABLE II
COMPARISON OF DYNAMIC ANALYSIS TOOLS

Tool	Precision	Time	OS	Gran.	Control
Valgrind	Sec. III	Slow	Multiple	Coarse	High
ltrace	Only III-A	Slow	Linux	Fine	Medium
RTLD-Audit	Only III-A	Fast	Linux/Solaris	Coarse	Low

binary, such as tracking only the library calls made within function `main`. RTLD-AUDIT, in contrast, offers very little control, as it also records library calls performed by the dynamic linker itself, which is not part of the program, and usually happen before and after the `main` function runs. An advantage of the version based on RTLD-AUDIT is its low running time, as discussed in Section V-C. Finally, LTRACE stands out for its granularity (**Gran.**), as it tracks both arguments and return values of function calls.

IV. A CODE CLASSIFICATION GAME

To investigate the effectiveness of the binary diffing approach proposed in this paper, Section V compares LIBSIG with methodologies of similar purpose. To perform this comparison, we use the *Game Framework* proposed by Damásio et al. [10, Def.2.4]. Damásio et al. has evaluated the effectiveness of binary analyzers (the so called "*classifiers*") by pitching them against code transformation tools (the so called "*evaders*") in four different adversarial games. Each game would differ on the amount of freedom given to the evader (it might or might not transform the program) and on the amount of information given to the classifier (it might or might not know which transformation the evader can use).

In Damásio et al.'s work, games consisted in code classification challenges: given a number of programming problems, and one program, the classifier should infer which problem the program solves. In this paper, we are rather interested in the challenge that VenkataKeerthy et al. [51] define as "*Binary Matching*" (see Definition 6.2 in their original presentation). Therefore, our Definition 3 combines Damásio et al.'s games with VenkataKeerthy et al.'s matching.

Definition 3 (Adversarial Game)**.** *A game on a set $B_n = \{b_1, b_2, \ldots, b_n\}$ of n binaries involves the following players:*

- *An* Evader *function $E : B_n \to B'_n$ such that, for each $1 \leq i \leq n$, the program $b'_i = E(b_i)$:*
  - *preserves the semantics of b_i (i.e., for the same inputs, b_i and b'_i produce the same outputs),*
  - *might differ in its syntax.*

  We call b'_i the transformed *version of b_i, and define $B'_n = E(B_n)$ as the set of all transformed programs.*
- *A* Classifier *function $C : B_n \times B'_n \to B_n \times B'_n$, which attempts to match each binary $b_i \in B_n$ to a corresponding binary $b'_j \in B'_n, 1 \leq i, j \leq n$.*
- *An* Arbiter *function $A : B_n \times B'_n \to \mathbb{N}$, which evaluates a matching by counting the number of* correct *matches. A match (b, b') is considered correct if b and b' implement the same semantics.*
- *The* Score *of the game between C and E is defined as:*

$$A(C(B_n, \mathit{shuffle}(E(B_n))))$$

where $\mathit{shuffle}(E(B_n))$ denotes a random permutation of the binaries in B_n.

Notice that Definition 3 intentionally leaves the notions of *Classifier*, *Evader*, and *Arbiter* vague, as these concepts can be instantiated in different ways. As a concrete example, Figure 7 illustrates Definition 3 with specific instantiations.

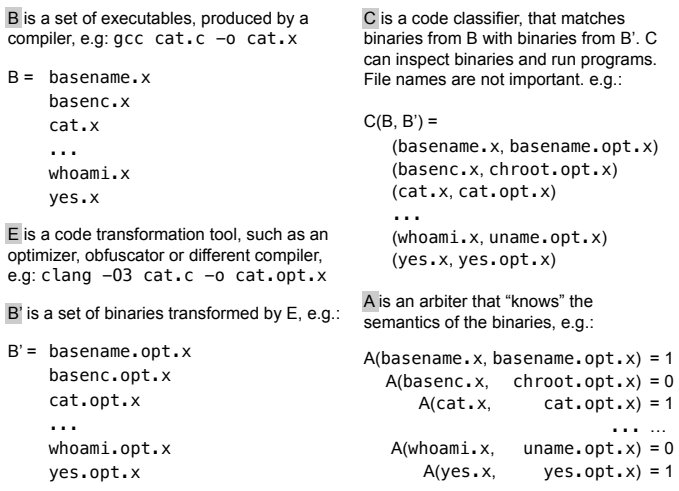

Fig. 7. A binary matching game using the programs in GNU Core Utilities. The score of the classifier is the sum of the number of correct matches.

Classifiers: The literature contains many examples of classifiers, which we categorize into two types:

- **Static:** The classifier is allowed to inspect the object file but cannot execute it.
- **Dynamic:** The classifier is allowed to inspect the binary file and observe one or more executions of it.

Section V evaluates both static and dynamic classifiers. Our LIBSIG classification engine, as well as ROUXINOL [16], are examples of dynamic classifiers. In contrast, tools such as SAFE [34] and BINDIFF [1] are examples of static classifiers.

Regardless of the approach, this paper imposes two restrictions on classifiers:

1) Classifiers do not have access to the source code of the binary (such as `cat.c` in Figure 7).
2) Classifiers do not have access neither to the source code of libraries, nor to their binary implementation; hence, cannot instrument or otherwise modify them.

Additionally, classifiers are allowed to match two or more binaries in B_n with the same binary in B'_n. This means that a trivial classifier that matches every binary in B_n to a single binary in B'_n will score exactly one point. This allowance for repeated matching is necessary because it reflects the behavior of some of the tools evaluated in Section V.

Evaders: An evader is any function that modifies the syntax of a program while preserving any visible output the program might produce. In contrast to classifiers, this paper assumes that evaders have access to the source code or programs; hence, they can recompile them. Evasion techniques evaluated in Section V include:

- Using a different compiler, e.g., `gcc` vs. `clang`;
- Using different versions of the same compiler, e.g., `clang 9.0` vs. `clang 16.0`;
- Applying different optimization levels, e.g., `clang -O0` vs. `clang -O3`;
- Applying obfuscators, such as OLLVM [25] or KHAOS [65].

V. EVALUATION

The goal of this section is to evaluate binary diffing based on matching of library signatures. To this end, we analyze the following research questions:

RQ1: How effective is LIBSIG when compared with state-of-the-art binary diffing tools in terms of accuracy?
RQ2: How does the LIBSIG compares with other binary diffing tools in terms of running time?
RQ3: What is the overhead that LIBSIG imposes onto the execution time of programs, and how does this overhead compare with that of similar tools?

To answer these questions, we extract library signatures in three different ways:

- LIBSIG-v: the version of LIBSIG implemented on top of VALGRIND.
- LIBSIG-r: the version of LIBSIG implemented on top of RTLD-AUDIT.
- LTRACE: the Linux tool, as described in https://man7.org/linux/man-pages/man1/ltrace.1.html.

Henceforth, we use LIBSIG to refer to the generic implementation of library signatures. When it is necessary to distinguish between the VALGRIND- and RTLD-AUDIT-based versions, we use the suffixes '-v' and '-r', respectively. Both LIBSIG-v and LTRACE track only library calls that originate from the program's `.text` segment. In contrast, LIBSIG-r also records library calls performed by the RTLD-AUDIT API itself, as well as calls made by any external module loaded alongside the target program. This occurs because RTLD-AUDIT does not expose the target addresses of intercepted calls, making it non-trivial to determine whether a call originated from the program's `.text` segment. We encountered this situation in the CHCON benchmark from COREUTILS, a utility that changes the SELINUX (Security-Enhanced Linux) context of a file. To perform this task, CHCON loads the `selinux.so` library. As a result, LIBSIG-r records additional calls originating from this library, which are not invoked by the program's own code. To ensure a consistent analysis using LIBSIG-r, we recompiled CHCON without linking against `selinux.so`.

A. Experimental Setup

Before we discuss each of the research questions, we present our experimental setup, including choice of benchmarks, classifiers and evaders.

1) On the Choice of Benchmarks: We run games on a set B_{102} formed by 102 programs taken from `CoreUtils`. `CoreUtils`, or GNU Core Utilities, is a collection of command-line tools in Unix-like operating systems. These tools, like `ls`, `cat`, `cp`, `mv`, and `rm`, are primarily written in C and adhere to POSIX standards, ensuring consistent behavior across diverse compliant systems. `CoreUtils` contains 106 executables; however, four of them could not be compiled with Khaos, and thus were removed from the evaluation. In order to make binary diffing harder, every program in B_{102} is stripped of unnecessary symbols to execute using `strip --strip-all`. External functions names are kept in the binary, since they are used to link to the correct implementation in the libraries. All binaries have the PLT preserved, otherwise LTRACE could not be compared against LIBSIG, which handles all the cases discussed in Section III-A.

2) On the Choice of Evaders: In this paper, the baseline set of binary files, e.g., B_{102}, is compiled with `clang` 9.0 at the `-O0` optimization level. Thus, according to Definition 3, an evader is any modification of this compilation pipeline. We build evaders by varying the compiler (`gcc` v13.1 and `clang`), or the version of the compiler (`clang` 9.0 and `clang` 16.0), or the optimization level (`-O0`, `-O1`, `-O2` and `-O3`), or by applying obfuscations on code. Evasion via obfuscation uses either OLLVM [25] or KHAOS [66], which apply one of the following obfuscation approaches[4]:

- **ollvm-bcf:** Bogus Control Flow inserts dead-code guarded by predicates that are difficult to elide statically.
- **ollvm-fla:** Control-Flow Flattening transforms the program into a big switch running inside a loop, where each basic block sets the case for the next basic block [30].
- **ollvm-sub:** Instruction Substitution replaces binary operators, like addition, subtraction or boolean operators, by functionally equivalent sequences of instructions.
- **ollvm-all:** The combination of the three obfuscation techniques (bcf, fla, sub) available in OLLVM.

[4]The distribution of `Khaos` provides users with five different configurations, which Zhang et al. call *Fussion*, *Fission*, *Hidden*, *Original* and *All* (See Figure 6 of Zhang et al.'s work [67]). We have been able to run KHAOS only with *Fission* and *Hidden* independently enabled. The other transformations caused compilation crashes or produced binaries that did not terminate successfully.

- **khaos-fis:** Fission splits a function into sub-functions. The division ensures that each extracted region has a single entry point (but potentially multiple exit points).
- **khaos-hid:** Hidden Flow replaces direct jumps with exception-handling branches. This mechanism is implemented via runtime lookups through exception tables.

Both these obfuscators are built on top of `clang` 9.0.

3) On the Choice of Classifiers: The different classifiers might approach binary diffing at two granularities:

- **Function:** BINDIFF, SAFE and ASM2VEC perform matching per function: upon receiving two binary files, b and b', they compare each function on each binary, for a quadratic number of comparisons. Given two sets of Binaries B_{102} and B'_{102}, we report a match between $b \in B_{102}$ and $b' \in B'_{102}$ for the two binaries with the closest distance between the sum of distances of functions.
- **File:** The other approaches use file embeddings, meaning that they do not distinguish individual functions within a binary file. These embeddings are histograms of opcodes that come from x86 instructions:
  - **S-x86:** (Static Binary View) Instructions in the `.text` section of binaries, produced via the `Capstone` Disassembler v5.0 [15].
  - **D-x86:** (Dynamic Binary View) Instructions observed via CFGGRIND [46], during one or more executions of a program. Each execution of the same instruction type is counted.
  - **H-x86:** (Hybrid, e.g., Dynamic Program Slice) Instructions observed via CFGGRIND [46], during one or more executions of a program. This is the control-flow graph of the program parts covered during execution. Thus, multiple executions of the same type of instructions are counted only once.

Distance: Except for BINDIFF, each classifier uses some notion of distance to match binaries:

- **Edit:** LIBSIG and LTRACE use the Edit Distance, ie., the minimum number of insertions or deletions necessary to convert one library signature into another.
- **Cosine:** The cosine of the angle between the vectors. Henceforth, we suffix tools that use this distance with **(C)**, e.g., S-X86 (C).
- **Euclidean:** The length of the line segment between the points formed by two histograms. We indicate it with the suffix **(E)**, e.g., **S-x86 (E)**.

The Cosine distance is recommended in ASM2VEC's original publication [11]; whereas the Euclidean distance is used in the tools that we took from the ROUXINOL framework (S-X86, D-X86, H-X86) [16]. BINDIFF does not use a concept of distance. Instead, it relies on an assortment of heuristics to match functions, including hash of instructions and graph-related metrics. More about it is available in the tool's documentation[5].

Setup: The baseline set B_n (e.g., B_{102}) of Definition 3 is produced with `clang` version 9.0. Whenever we mention `clang` as an evader, we refer to version 16.0. For instance, Tables III and IV use `clang` 16.0 at different optimization levels to yield the rows labeled `clang-O0`, `clang-O1`, etc.

Input Selection: For analyses, such as LTRACE, LIBSIG, D-X86, and H-X86, that require the dynamic behavior of programs, a representative input must be selected. To this effect, the baseline compiler was used, without optimizations, to run the entire `CoreUtils` test suite in order to identify, for each program, the input for which LIBSIG-v produces the longest library signature. These selected inputs were then used to run the evaluation games. Instead of selecting a single input per program, one could concatenate the signatures produced by all available inputs, for instance. However, we chose to use only the input that yields the longest signature to simplify the construction of a reproducible scientific artifact.

B. Accuracy of LIBSIG *Compared with Previous Work*

This section compares the *accuracy* of the different classifiers, when pitched against different evaders. Accuracy is measured via the *Arbiter* of Definition 3; hence, varying from 0 (no correct match) to 102 (every binary is correctly matched).

Discussion: Table III shows how LIBSIG compares with different binary diffing tools. The matching of library signatures via LIBSIG or LTRACE is the most precise binary diffing approach. In every experiment, these tools only failed to achieve a full matching score due to either executables with very short library signatures or due to compiler optimizations. In this case, compilers such as `clang` and `gcc` can modify some library calls. Below we have a non-exhaustive list of examples:

- Replacing sequences of `malloc` and `memset` with `calloc`
- Replacing `atoi` with `strtol` for error handling.
- Replacing `sprintf` with `snprintf` for safety.
- Constant-folding calls to `strlen`, when applied onto string literals.
- Replacing calls to `strcpy` with `stncpy` or `memcpy`.
- Performing some form of strength-reduction on calls like `pow`(x, c), where c is a constant.
- Replacing calls to `printf` on constant strings with `puts` or `fwrite`.
- Vectorizing calls to `memcmp` at `-O2` or higher.
- Replacing calls to `abs` with bitwise operations.

A most appaling example of such modifications was observed in `numfmt`. The library signature of the non-optimized binary contains 100,055 calls, whereas once compiled with `clang -O1` (or higher), it contains only 74. In this case, `clang` was able to optimize the code by removing almost 100 thousand calls to `putchar` that were being used for printing alignments.

Another example of aggressive optimizations takes place in the `readlinebuffer_delim`: a function that is shared amongst some `CoreUtils` programs, such as `uniq` and `comm`. This function reads lines, one byte at a time, from a file until a delimiter is found using `libc`'s `getc`. An optimizer may replace the sequential function `getc` with a function like `__uflow` that uses buffered I/O operations. For programs such as `uniq`, this means a reduction from 1,000 calls to `getc`,

[5]The different heuristics used by BINDIFF are described at https://github.com/google/bindiff/blob/main/docs/concepts.md

in the unoptimized version, to as few as 2 calls to `__uflow`, in the optimized one. This drastic reduction has consequences when running the game in programs with little variety of external calls. Consider the game for `uniq` with LIBSIG: the 1,042 total calls in the unoptimized version (baseline) drops to 52 calls in the optimized version (`clang-O1`), yielding a match of only 7.86%. To put that in perspective, the next closest match was `comm` with 7.71% (a program that also uses `readlinebuffer_delim`). However, for such small difference, LTRACE or RTLD-AUDIT fail to correctly match `uniq` for evaders `clang-O1`, `clang-O2` and `clang-O3` in Table III. This is the consequence of missed calls that VALGRIND is able to track (as discussed in §III-B).

C. Runtime Comparison

This section compares LIBSIG in terms of *extraction* time and *matching* time with other binary diffing tools. Extraction time is the time to convert programs to a format that can be compared (sequences in the case of LIBSIG and LTRACE, and vectors in the other cases). Matching is the time of comparing these embeddings. Notice that, given the nature of the game described in Definition 3, matching, in this experiment, involves 102×102 comparisons.

Discussion: Table IV reports our observations. We open this analysis by emphasizing that our impression is that none of the binary matching tools has been engineered to run fast. Thus, running times, be extraction or matching, tend to be slow. As an example, the ROUXINOL-based tools (S-X86, H-X86 and D-X86, all implemented in Python) do not keep embeddings in memory: rather, they read and close files whenever necessary to compare two binaries. BINDIFF, in turn, runs on the JVM, and pays its cost in extraction and matching. The former is particularly slow, as it also relies on GHIDRA [14] to represent and analyze binary programs.

The `CoreUtils` programs run for a very short time. Thus, in spite of VALGRIND's heavy overhead (to be analyzed in Section V-D), the extraction of embeddings, be it via CFGGRIND, for the ROUXINOL-based tools, be it via LIBSIG, is relatively fast, compared to the time that tools such as BINDIFF, ASM2VEC or SAFE take to produce embeddings. Matching is also faster, sometimes by an order of magnitude, when using library signatures which use the same matching infrastructure. However, we emphasize that this infrastructure was engineered to do well on the game proposed in this experiment.

D. Runtime Overhead

This section analyzes the overhead of LIBSIG, either implemented via VALGRIND or via RTLD-AUDIT, and contrasts it with the overhead of other tools. We shall be reporting *relative* execution times; that is, the running time difference of executing a program with and without the intervention of dynamic analysis.

Discussion: Figure 8 compares the runtime overhead of four different dynamic analyzers: NULGRIND, LIBSIG (either via VALGRIND or via RTLD-AUDIT), LTRACE, and CFGGRIND. All of these tools, except for LTRACE, are plugins built on top of the VALGRIND framework. NULGRIND is a minimal VALGRIND plugin designed primarily for testing, debugging, and developing new VALGRIND-based tools. Unlike analyzers such as MEMCHECK, CALLGRIND, CFGGRIND, or LIBSIG, which perform extensive instrumentation, NULGRIND performs no program analysis beyond basic supervision of the execution.

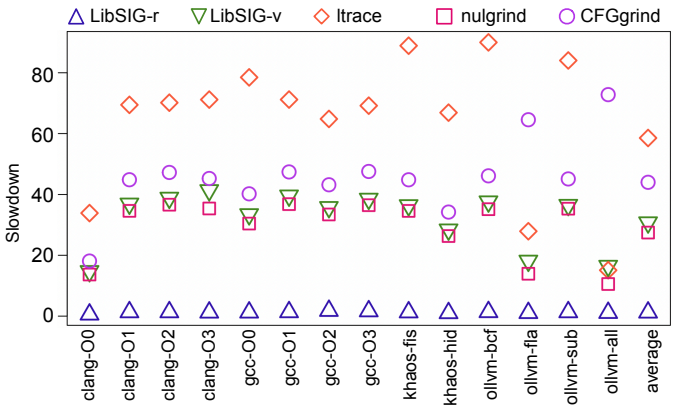

Fig. 8. Slowdown of different evaders across analysis tools. The "average" in the last column is the geometric mean.

LIBSIG VS LTRACE: LTRACE is the slowest tool, introducing a slowdown of nearly 60× relative to native execution. This is expected, as LTRACE records parameters and return values. LIBSIG-v, which tracks only call addresses, is significantly faster, with overheads remaining below 30× in most cases. In principle, LTRACE could be modified to track less information, but the effort might not be trivial because its additional features (e.g., parameter tracking) are intrinsic to its implementation. Nevertheless, this gap between LIBSIG and LTRACE may be reduced depending on the ratio of visible to invisible instructions in the program [69]. For example, `ollvm-fla` and `ollvm-all` are performing control flow flattening that changes this balance. More instructions are being executed than libraries calls are being called, thus LIBSIG and LTRACE performance is similar. Finally, LIBSIG-r is the fastest tool, adding an average overhead of 25%: the original benchmarks run in 1.43 seconds on average, and on 1.77 seconds once RTLD-AUDIT is active.

LIBSIG vs other VALGRIND tools: CFGGRIND incurs higher overhead than LIBSIG, as expected, since it tracks every instruction to construct a dynamic control-flow graph. Despite this, it remains faster than LTRACE, except when imbalances arise when the relation of visible to invisible instructions is changed by obfuscations such as control flow flattening. As expected, NULGRIND introduces the lowest overhead, performing no instrumentation beyond basic emulation. Interestingly, the gap between NULGRIND and LIBSIG is minimal, indicating that LIBSIG's instrumentation – of simply detecting jumps outside the `.text` section – is extremely lightweight. Most of its runtime cost comes not from instrumentation itself, but from the inherent overhead of VALGRIND's execution environment.

TABLE III
COMPARISON OF DIFFERENT EVADERS ACROSS ANALYSIS TOOLS AND SIMILARITY METRICS. WE LET (C) DENOTE THE COSINE DISTANCE AND (E) DENOTE THE EUCLIDEAN DISTANCE.

Evader	LIBSIG-r	LIBSIG-v	LTRACE	BINDIFF	SAFE (C)	S-X86 (C)	S-X86 (E)	ASM2VEC (C)	H-X86 (C)	H-X86 (E)	D-X86 (C)	D-X86 (E)
clang-O0	102	102	102	97	87	20	10	56	49	51	65	64
clang-O1	90	91	90	20	5	1	1	1	2	16	3	56
clang-O2	89	90	89	14	5	1	1	1	1	13	4	55
clang-O3	89	90	89	11	3	1	1	1	1	14	4	55
gcc-O0	102	102	102	85	36	5	27	1	33	47	30	61
gcc-O1	93	93	93	26	9	4	1	1	1	13	4	54
gcc-O2	90	91	90	13	6	3	1	0	4	16	3	54
gcc-O3	90	91	90	8	6	3	1	0	3	13	3	45
khaos-fis	102	102	102	86	46	19	1	35	11	40	35	62
khaos-hid	102	102	102	75	4	1	1	3	7	33	26	61
ollvm-bcf	102	102	102	76	2	2	1	2	1	7	5	70
ollvm-fla	102	102	102	83	9	2	1	7	3	10	2	39
ollvm-sub	102	102	102	96	85	14	6	64	59	68	67	91
ollvm-all	102	102	102	9	2	2	1	3	1	3	2	21

TABLE IV
RUNTIME OVERHEAD IN SECONDS FOR FEATURE EXTRACTION AND MATCHING.

Time (s)	LIBSIG-r	LIBSIG-v	LTRACE	BINDIFF	SAFE (C)	S-X86 (C)	S-X86 (E)	ASM2VEC (C)	H-X86 (C)	H-X86 (E)	D-X86 (C)	D-X86 (E)
Extraction	28	654	1,187	37,885	156,466	88	88	61,015	1,318	1,318	1,318	1,318
Matching	3,375	3,452	3,328	10,887	26,409	1,562	1,564	25,670	3,669	3,697	3,678	3,666
Total	3,403	4,106	4,515	48,772	182,875	1,650	1,652	86,685	4,987	5,015	4,996	4,984

E. Discussion: Limitations of LIBSIG

The results in Section V-B indicate that LIBSIG is very effective as a code matching technique. However, it does have limitations, some of which we discuss below:

- **Hindsight:** LIBSIG was tested against evaders that were not engineered to alter library calls. Except for `clang/gcc` at high optimization levels, none of the program transformation tools touches these calls. And even these two compilers only modify them marginally. However, it would be relatively easy to implement an obfuscator that adds effect-free library calls to programs; hence, modifying the library signature of those programs.
- **Length:** Short library signatures tend to lead to more collisions. As an example, every `CoreUtils` program reads a `--version` flag, which prints release data. In most cases, this flag produces a library signature with less than ten calls. Thus, if we compare programs using exclusively this flag as the input, then several of them will have the same library signature.
- **Library Substitutions:** As seen in Section V-B, both `gcc` and `clang` might change particular library calls at high optimization levels. Additionally, there are obfuscation techniques, such as Omar et al. [40]'s, which might insert library calls into sequential code to make it run in parallel. Substitution of library calls could also be performed by the loader, e.g., replacing `strlen` to `strlen_sse2`. In our experiments, we only observed substitutions due to compiler optimizations, with very limited effect; however, they could impact evaluations involving short signatures.

VI. RELATED WORK

This paper observes that a binary fingerprint based on a program's library signature resists common obfuscation techniques significantly better than other classification approaches, including recent ones. To the best of the authors' knowledge, this observation is novel. However, prior work has explored the use of library calls as auxiliary features for code classification, as well as techniques for obfuscating such calls. This section revisits that body of literature.

A. Library-Aware Classifiers

Binary similarity analysis has attracted research attention due to its practical importance[6]. Since program equivalence is undecidable [44], binary similarity relies on heuristics. Early approaches operated on source code [22], but research shifted toward binary code in the 1990s. Initial efforts focused on sequence alignment algorithms [3, 8] and hash functions [55], techniques still in use today [7, 20, 31, 34, 53]. However, modern research predominantly explores machine learning–based methods [11, 12, 31, 34, 54, 62].

Use of Library Calls in Static Analyzers: There exist previous work that incorporates library calls to enhance binary classification. For instance, Kotov and Wojnowicz [28] and Lakhotia et al. [29] statically identify obfuscated API calls by looking for suspicious patterns in the target binary, such as function arguments. As another example, Ikram et al. [23] extract a graph of library calls from a binary, and use it as that binary's signature. More recently, VenkataKeerthy et al. [51] augment their program embedding, VEXIR2VEC, with library calls. By incorporating incorporating library calls, VenkataKeerthy et al. observed an improvement in accuracy, from 63% to about 81%. However, their approach remains vulnerable to obfuscation: precision drops from 81% to 27% when analyzing

[6]Reflecting this diversity, the *Awesome Binary Similarity* webpage catalogs approximately 214 publications as of June 2024—https://github.com/SystemSecurityStorm/Awesome-Binary-Similarity.

OLLVM-obfuscated binaries. Similarly, BINFINDER and SAFE use library calls as comparison markers, although both lag behind VEXIR2VEC in terms of precision. These methods are purely static. While they detect function calls, they cannot determine call order. The obfuscation techniques that we discuss in Section VI-B effectively evade static analyses but, as seen in Section II, fail against dynamic approaches.

Use of Library Calls in Dynamic Analyzers: The idea of using a trace of library calls to detect malware is not new. This possibility was first demonstrated by Kolbitsch et al. [27], and then by Patanaik et al. [41]. Kolbitsch et al. showed how to use the data dependencies between function calls to build a signature for a binary. Subsequently, Patanaik et al. showed that the sequence of kernel calls performed by the `Bagle.A` virus remain the same under different obfuscation methodologies. In contrast to this paper, this body of work focuses exclusively on kernel calls, not including library calls, such as `libc`'s. They also try to match subgraphs of the "*dynamic call graph*" of a program; hence, the ordering of calls is less important: separate sequences of interdependent calls can be matched in any order. Our work, in turn, matches sequences of function invocations (the library signature); thus, ordering is fundamental.

B. Defeating Library-Aware Static Classifiers

Several obfuscation techniques exist to hide library calls in binaries, but most traditional approaches remain vulnerable to dynamic analysis. For example:

- **Call redirection** [6] replaces library calls with trampoline jumps, but dynamic analysis (including LIBSIG) can still observe the final jump to the destination API.
- **Position obfuscation** [49] relocates API prologues to user space, yet the subsequent jumps back to library code remain detectable during execution.
- **Call site tampering** [47] employs runtime API resolution (e.g., via hash lookups), but ultimately still reveals library calls during execution.
- **Dynamic code loading** [26] stores part of a program in encrypted form. This code is decrypted during runtime using a key hidden somewhere in the app. However the deciphered jump target is visible at runtime.

An example of call site tampering is the obfuscator of Mather [35], which replaces symbol references with hash codes and uses stubs for dynamic resolution for ELF binaries in Linux systems. As an example of dynamic code loading, Nguyen [39] removes dynamic names from the symbols table of Mach-O binaries and restores them at runtime through either an external library or injected code. Nevertheless, while these methods effectively thwart static analysis, they remain susceptible to dynamic analysis like our LIBSIG system. The fundamental limitation is that the binary must eventually transfer control to the actual library functions; hence, leaving the code's `.text` section. Our approach monitors low-level jumps to detect when the boundary between local and library code is crossed during program execution.

C. Defeating Library-Aware Dynamic Classifiers

There are obfuscation techniques capable of defeating dynamic analyses such as LIBSIG, although they typically require privileged access to specific computational resources. For example, if an attacker is permitted to load a malicious kernel module, Srivastava et al. [48]'s *Illusion Attack* enables the redirection of malicious operations into seemingly benign `ioctl` buffers. The tampered kernel module can then unmarshal and execute the actual privileged operations. To external observers, only benign `ioctl` calls are visible, effectively concealing the malicious behavior.

More recently, Li et al. [32] introduced an obfuscation technique, APIASO, that could evade dynamic analyses like the one proposed in this paper. APIASO extracts and encrypts key functions from Windows APIs and relocates them into the user code space. At runtime, the program executes these relocated functions instead of calling the original APIs in system DLLs, such as `kernel32.dll`. A limitation of this approach is that it requires recompiling the source code with access to the target library code. Consequently, the software may become incompatible with updated versions of the operating system or other environments where the library interface has changed.

VII. CONCLUSION

This paper introduced a novel approach to binary diffing based on the concept of *Library Signature*, which captures the sequence of external library calls made during a program's execution. This simple behavioral fingerprint remains resilient against a wide range of binary transformations, including different compilers, optimization levels, and common obfuscation techniques such as OLLVM and KHAOS. This resilience stands in contrast to other binary matching techniques, including ASM2VEC, SAFE, BINDIFF, and ROUXINOL, whose precision degrades sharply under adversarial conditions.

This paper demonstrates that LIBSIG is effective against current obfuscation techniques; however, the limitations discussed in Section V-E open avenues for future work. First, it is straightforward to envision evasion strategies that alter a program's library signature, for instance, by inserting innocuous calls or replacing existing calls with semantically similar ones. As is often the case in software security, this problem resembles an arms race. Should such evasion techniques become widespread, LIBSIG could be extended with countermeasures such as clustering semantically equivalent calls, filtering for calls with observable side effects, or modeling temporal and causal relationships (e.g., tracking the expected pairing of `malloc` and `free`). Second, LIBSIG has not yet been tested against large, complex applications. Although we believe it would scale well to larger binaries, this evaluation remains an important direction for future empirical study.

Software: LIBSIG is available in Zenodo [45]. An up-to-date version is available at https://github.com/rimsa/LibSIG.

ACKNOWLEDGMENT

This project was supported by FAPEMIG (APQ-00440-23), CNPq (#444127/2024-0) and CAPES (PRINT).

APPENDIX

A. Abstract

This artifact reproduces the experiments conducted in Section V. A docker container with scripts to rebuild the paper results, perform the game and produce Figure 8 and Tables III and IV automatically can be found in https://doi.org/10.5281/zenodo.17082032. The experiments are separated in three stages. Stage 1 performs the runtime evaluation of CoreUtils executing the baseline (no simulation), LIBSIG RTLD version and the VALGRIND's plugin version, LTRACE, NULGRIND (VALGRIND with no instrumentation), and CFGGRIND for all evaders (§V-A2). Stage 2 performs the evaluation of BINDIFF. Stage 3 performs the evaluation of ASM2VEC, SAFE, x86 histograms and CFGGRIND.

Additionally, the artifact contains a demonstration script to showcase the claims made in Section II regarding the behavior of LIBSIG and LTRACE on Example 1 when different linking strategies are employed, such as eager linking (§III-A1), lazy linking (§III-A2), and runtime linking (§III-A3).

B. Artifact check-list

- **Program:** VALGRIND with LIBSIG and CFGGRIND, CoreUtils v9.6, Obfuscator-LLVM, Khaos, GHidra, ASM2VEC and SAFE.
- **Compilation:** clang-9, clang-16 and gcc-10.
- **Binary:** CoreUtils compiled binaries for all evaders. This will be produced by the artifact scripts.
- **Run-time environment:** Any operating system that supports Docker. The docker container uses Linux with an x86-64 architecture.
- **Hardware:** Any hardware that supports docker.
- **Metrics:** Accuracy compared to previous work (§V-B), runtime comparison (§V-C) and runtime overhead (§V-D).
- **Output:** Figure 8 and Tables III and IV that are generated automatically.
- **Experiments:** Rebuild the results of the paper, perform the game and produce the figures and tables.
- **How much disk space required:** The docker image requires ∼20.5Gb and the dataset requires ∼4.5Gb.
- **How much time is needed to prepare workflow:** To build the docker image requires ∼4 hours.
- **How much time is needed to complete experiments:** To run the experiments, we have observed the following times on a typical 3.2GHz machine when using Docker: ∼13 hours for stage 1, ∼6 hours for stage 2, and ∼15 days for stage 3.
- **Publicly available:** LIBSIG is publicly available at https://github.com/rimsa/LibSIG and the artifact is available at https://doi.org/10.5281/zenodo.17082032 [45].
- **Workflow framework used:** Docker.

C. Description

1) How delivered: Delivered via Docker, available through Zenodo [45] (https://doi.org/10.5281/zenodo.17082032).

D. Installation

1) Download the artifacts from Zenodo [45] (https://doi.org/10.5281/zenodo.17082032) and decompress it:
```
$ tar -jxvf libsig-artifacts.tar.bz2
```
2) Choose one of the following options:
   - Use the dataset provided from our experiments. Decompress it by keeping the same user id used by docker.
```
$ sudo tar --same-owner \
    -jxvf dataset.tar.bz2
```
   - Or, generate the dataset from scratch.
```
$ mkdir dataset
$ chmod 777 dataset
```
3) Choose one of the following options:
   - Build the image.
```
$ docker build --platform=linux/amd64 \
    -t libsig_image .
```
   - Or, use a prebuilt image.
```
$ bunzip2 libsig_image.tar.bz2
$ docker load -i libsig_image.tar
```
4) Run the container.
```
$ docker run --rm -it --name libsig_$$ \
    -v ./dataset:/work/dataset libsig_image
```

E. Experiment workflow

From within the docker container terminal, you can choose between the following commands.

- `demo`: Perform a step-by-step demonstration of LIBSIG.
- `rebuild`: Execute scripts to rebuild the results from scratch. You can choose to rebuild a specific stage, for example: `rebuild stage1`.
- `game`: Execute scripts to perform the game from scratch. You can choose to game a specific stage, for example: `game stage1`.
- `products`: Execute scripts to generate the products of the paper (figures and tables). You can choose to generate products for specific stage, for example: `products stage1`.

If you do not wish to perform each command (`rebuild`, `game`, `products`) separately, you can choose to run everything at once with the following command.

- `everything`: Execute `rebuild`, `game`, and `products` for each of the three stages to perform the whole analysis.

Notice that every command is independent of each other. For example, an evaluator may choose to run only the game on the data we collected in our experiments by choosing the command `game`, or `game stage1` to game only the first stage. If one wishes to regenerate the entire dataset from scratch, use the `everything` command which will run the `rebuild`, `game` and `products` for each stage at a time. Thus, one will be able to evaluate the products of each stage as it is completed.

F. Evaluation and expected result

After completion, everything produced can be found in the following directories:

- `dataset/binaries`: the binaries generated for all evaders.
- `dataset/results`: all intermediate files produced from the rebuilding process.
- `dataset/games`: all intermediate files produced from the gaming process.
- `dataset/products`: Figure 8 and Tables III and IV.

REFERENCES

[1] Thomas Dullien (a.k.a. Halvar Flake). Bindiff, 2011. URL https://github.com/google/bindiff. Accessed: 2025-03-14.

[2] Priya Arora, Rashmi Gupta, Nidhi Malik, and Anil Kumar. Malware analysis types & techniques: A survey. In *ICIMMI*, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400709418. doi: 10.1145/3647444.3652439. URL https://doi.org/10.1145/3647444.3652439.

[3] Brenda S. Baker and Udi Manber. Deducing similarities in java sources from bytecodes. In *1998 USENIX Annual Technical Conference (USENIX ATC 98)*, New Orleans, LA, June 1998. USENIX Association. URL https://www.usenix.

org/conference/1998-usenix-annual-technical-conference/deducing-similarities-java-sources-bytecodes.

[4] Martial Bourquin, Andy King, and Edward Robbins. Binslayer: accurate comparison of binary executables. In *PPREW*, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318570. doi: 10.1145/2430553.2430557. URL https://doi.org/10.1145/2430553.2430557.

[5] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004. URL https://dspace.mit.edu/handle/1721.1/16674.

[6] Binlin Cheng, Jiang Ming, Erika A Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean-Yves Marion. Obfuscation-resilient executable payload extraction from packed malware. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3451–3468, 2021.

[7] Josh Collyer, Tim Watson, and Iain Phillips. Faser: Binary code similarity search through the use of intermediate representations, 2023.

[8] Krish Coppieters. A cross-platform binary diff. https://www.drdobbs.com/embedded-systems/a-cross-platform-binary-diff/184409550, 1995. [Online; accessed 12-Nov-2023].

[9] Anderson Faustino da Silva, Edson Borin, Fernando Magno Quintão Pereira, Nilton Luiz Queiroz Junior, and Otávio Oliveira Napoli. Program representations for predictive compilation: State of affairs in the early 20's. *J. Comput. Lang.*, 73:101171, 2022. doi: 10.1016/j.cola.2022.101171. URL https://doi.org/10.1016/j.cola.2022.101171.

[10] Thaís Damásio, Michael Canesche, Vinícius Pacheco, Marcus Botacin, Anderson Faustino da Silva, and Fernando M. Quintão Pereira. A game-based framework to compare program classifiers and evaders. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2023, page 108–121, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701016. doi: 10.1145/3579990.3580012. URL https://doi.org/10.1145/3579990.3580012.

[11] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *S&P*, pages 472–489, USA, 2019. IEEE Computer Society. doi: 10.1109/SP.2019.00003.

[12] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. DeepBinDiff: Learning program-wide code representations for binary diffing. In *NDSS*, pages 1–10, Reston, VA, USA, 2020. Internet Society (ISOC).

[13] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications*, pages 1–8, France, 2005. SSTIC Association.

[14] Chris Eagle and Kara Nance. *The Ghidra Book: The Definitive Guide*. no starch press, 2020.

[15] Nguyen Anh Quynh et al. Capstone engine. https://www.capstone-engine.org/, 2024.

[16] Anderson Faustino da Silva, Jeronimo Castrillon, and Fernando Magno Quintão Pereira. A comparative study on the accuracy and the speed of static and dynamic program classifiers. In *CC*, page 13–24, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714078. doi: 10.1145/3708493.3712680. URL https://doi.org/10.1145/3708493.3712680.

[17] Halvar Flake. Structural comparison of executable objects. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Lecture Notes in Computer Science, pages 161–173, Heidelberg, Germany, 2004. Springer. ISBN 3-88579-375-X.

[18] Free Software Foundation. Gnu general public license, version 3, 2007. URL https://www.gnu.org/licenses/gpl-3.0.html. Accessed: 2025-05-02.

[19] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ACM Comput. Surv.*, 54(3), April 2021. ISSN 0360-0300. doi: 10.1145/3446371. URL https://doi.org/10.1145/3446371.

[20] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection. In *Security*. USENIX, 2024.

[21] Wei Huang et al. Binsim: Binary similarity measurement for detecting software vulnerabilities. In *ICSE*, pages 495–504, New York, USA, 2011. IEEE Press. doi: 10.1145/1985793.1985856.

[22] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20 (5):350–353, may 1977. ISSN 0001-0782. doi: 10.1145/359581.359603. URL https://doi.org/10.1145/359581.359603.

[23] Muhammad Ikram, Pierrick Beaume, and Mohamed Ali Kaafar. Dadidroid: An obfuscation resilient tool for detecting android malware via weighted directed call graph modelling, 2019. URL https://arxiv.org/abs/1905.09136.

[24] Jacobsen v. Katzer. Jacobsen v. katzer, 535 f.3d 1373 (fed. cir. 2008). https://casetext.com/case/jacobsen-v-katzer, 2008. United States Court of Appeals for the Federal Circuit, Decided August 13, 2008.

[25] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm – software protection for the masses. In *SPRO*, pages 3–9, Washington, DC, US, 2015. IEEE.

[26] Ulf Kargén, Noah Mauthe, and Nahid Shahmehri. Characterizing the use of code obfuscation in malicious and benign android apps. In *ARES*, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400707728. doi: 10.1145/3600160.3600194. URL https://doi.org/10.1145/3600160.3600194.

[27] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *SSYM*, page 351–366, USA, 2009. USENIX Association.

[28] Vadim Kotov and Michael Wojnowicz. Towards generic deobfuscation of windows api calls, 2020. URL https://arxiv.org/abs/1802.04466.

[29] Arun Lakhotia, Eric Uday Kumar, and Michael Venable. A method for detecting obfuscated calls in malicious binaries. *IEEE Trans. Softw. Eng.*, 31(11):955–968, November 2005. ISSN 0098-5589. doi: 10.1109/TSE.2005.120. URL https://doi.org/10.1109/TSE.2005.120.

[30] Tımea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30(1):3–19, 2009.

[31] Xuezixiang Li, Yu Qu, and Heng Yin. Palmtree: Learning an assembly language model for instruction embedding. In *CCS*, page 3236–3251, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384544. doi: 10.1145/3460120.3484587. URL https://doi.org/10.1145/3460120.3484587.

[32] Yang Li, Fei Kang, Hui Shu, Xiaobing Xiong, Yuntian Zhao, and Rongbo Sun. Apiaso: A novel api call obfuscation technique based on address space obscurity. *Applied Sciences*, 13(16), 2023. ISSN 2076-3417. doi: 10.3390/app13169056. URL https://www.mdpi.com/2076-3417/13/16/9056.

[33] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 190–200. ACM, 2005. doi: 10.1145/1065010.1065034. URL https://doi.org/10.1145/1065010.1065034.

[34] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni,

Leonardo Querzoni, and Roberto Baldoni. SAFE: Self-attentive function embeddings for binary similarity, 2019. URL https://arxiv.org/abs/1811.05296.

[35] Keiran Mather. Obfuscating linux symbols: a novel approach to evade static analysis in linux malware, 2024. Available at https://www.bulletproof.co.uk/blog/tech-talk-obfuscating-linux-symbols on April 24th, 2025.

[36] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. Binsim: trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Security Symposium*, SEC'17, page 253–270, USA, 2017. USENIX Association. ISBN 9781931971409.

[37] Ibrahim Nadir, Haroon Mahmood, and Ghalib Asadullah. A taxonomy of iot firmware security and principal firmware analysis techniques. *Int. J. Crit. Infrastruct. Prot.*, 38(C), September 2022. ISSN 1874-5482. doi: 10.1016/j.ijcip.2022.100552. URL https://doi.org/10.1016/j.ijcip.2022.100552.

[38] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936332. doi: 10.1145/1250734.1250746. URL https://doi.org/10.1145/1250734.1250746.

[39] Anh Khoa Nguyen. Obfuscate api calls in mach-o binary, 2024.

[40] Rasha Salah Omar, Ahmed El-Mahdy, and Erven Rohou. Thread-based obfuscation through control-flow mangling, 2013. URL https://arxiv.org/abs/1311.0044.

[41] Chinmaya Kumar Patanaik, Ferdous A. Barbhuiya, and Sukumar Nandi. Obfuscated malware detection using api call dependency. In *SecurIT*, page 185–193, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450318228. doi: 10.1145/2490428.2490454. URL https://doi.org/10.1145/2490428.2490454.

[42] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *S&P*, page 709–724, USA, 2015. IEEE Computer Society. ISBN 9781467369497. doi: 10.1109/SP.2015.49. URL https://doi.org/10.1109/SP.2015.49.

[43] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. Unleashing the hidden power of compiler optimization on binary code difference: an empirical study. In *PLDI*, page 142–157, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454035. URL https://doi.org/10.1145/3453483.3454035.

[44] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. ISSN 00029947. URL http://www.jstor.org/stable/1990888.

[45] Andrei Rimsa. Libsig's artifacts, September 2025. URL https://doi.org/10.5281/zenodo.17082032.

[46] Andrei Rimsa, José Nelson Amaral, and Fernando M. Q. Pereira. Practical dynamic reconstruction of control flow graphs. *Softw. Pract. Exp.*, 51(2):353–384, 2021. doi: 10.1002/spe.2907. URL https://doi.org/10.1002/spe.2907.

[47] Kevin A. Roundy and Barton P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.*, 46(1), July 2013. ISSN 0360-0300. doi: 10.1145/2522968.2522972. URL https://doi.org/10.1145/2522968.2522972.

[48] Abhinav Srivastava, Andrea Lanzi, and Jonathon Giffin. System call api obfuscation (extended abstract). In *RAID*, RAID '08, page 421–422, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 9783540874027.

[49] Masaki Suenaga. A museum of API obfuscation on Win32. Technical report, Symantec Security Response, Tempe, AZ, USA, 2009. URL https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/a-museum-of-api-obfuscation-on-win32-en.pdf.

[50] PaX Team. Address space layout randomization (aslr), 2003. URL https://pax.grsecurity.net/docs/aslr.txt. Accessed: 2025-03-15.

[51] S. VenkataKeerthy, Soumya Banerjee, Sayan Dey, Yashas Andaluri, Raghul PS, Subrahmanyam Kalyanasundaram, Fernando Magno Quintão Pereira, and Ramakrishna Upadrasta. Vexir2vec: An architecture-neutral embedding framework for binary similarity, March 2025. ISSN 1049-331X. URL https://doi.org/10.1145/3721481. Just Accepted.

[52] Versata v. Ameriprise. Versata software, inc. v. ameriprise financial, inc., 2014 u.s. dist. lexis 166695 (e.d. tex. dec. 1, 2014). https://casetext.com/case/versata-software-inc-v-ameriprise-financial-inc, 2014. United States District Court for the Eastern District of Texas, Decided December 1, 2014.

[53] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. Jtrans: Jump-aware transformer for binary code similarity detection. In *ISSTA*, page 1–13, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393799. doi: 10.1145/3533767.3534367. URL https://doi.org/10.1145/3533767.3534367.

[54] Huaijin Wang, Pingchuan Ma, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. sem2vec: Semantics-aware assembly tracelet embedding. *ACM Trans. Softw. Eng. Methodol.*, 32(4), may 2023. ISSN 1049-331X. doi: 10.1145/3569933. URL https://doi.org/10.1145/3569933.

[55] Zheng Wang, Ken Pierce, and Scott McFarling. Bmat-a binary matching tool for stale profile propagation. *The Journal of Instruction-Level Parallelism*, 2:1–20, 2000.

[56] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *CCS*, page 363–376, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. doi: 10.1145/3133956.3134018. URL https://doi.org/10.1145/3133956.3134018.

[57] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. Patch based vulnerability matching for binary programs. In *ISSTA*, page 376–387, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380089. doi: 10.1145/3395363.3397361. URL https://doi.org/10.1145/3395363.3397361.

[58] Shouguo Yang, Long Cheng, Yicheng Zeng, Zhe Lang, Hongsong Zhu, and Zhiqiang Shi. Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection. In *DSN*, page 224–236, New York, USA, June 2021. IEEE. doi: 10.1109/dsn48987.2021.00036. URL http://dx.doi.org/10.1109/DSN48987.2021.00036.

[59] Shouguo Yang, Chaopeng Dong, Yang Xiao, Yiran Cheng, Zhiqiang Shi, Zhi Li, and Limin Sun. Asteria-pro: Enhancing deep learning-based binary code similarity detection by incorporating domain knowledge. *ACM Trans. Softw. Eng. Methodol.*, 33(1), November 2023. ISSN 1049-331X. doi: 10.1145/3604611. URL https://doi.org/10.1145/3604611.

[60] Shouguo Yang, Zhengzi Xu, Yang Xiao, Zhe Lang, Wei Tang, Yang Liu, Zhiqiang Shi, Hong Li, and Limin Sun. Towards practical binary code similarity detection: Vulnerability verification via patch semantic analysis. *ACM Trans. Softw. Eng. Methodol.*, 32(6), September 2023. ISSN 1049-331X. doi: 10.1145/3604608. URL https://doi.org/10.1145/3604608.

[61] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. Neural detection of semantic code clones via tree-based convolution. In *ICPC*, page 70–80, New York, USA, 2019. IEEE Press. doi: 10.1109/ICPC.2019.00021. URL https://doi.org/10.1109/ICPC.2019.00021.

[62] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. *AAAI*, 34(01):1145–1152, Apr. 2020. doi: 10.1609/aaai.v34i01.5466. URL https://ojs.aaai.org/

index.php/AAAI/article/view/5466.

[63] Hang Zhang et al. Malware detection via machine learning based on binary code analysis. In *ICSME*, pages 358–367, New York, USA, 2019. IEEE Press. doi: 10.1109/ICSME.2019.00049.

[64] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *ICSE*, page 783–794, New York, USA, 2019. IEEE Press. doi: 10.1109/ICSE.2019.00086. URL https://doi.org/10.1109/ICSE.2019.00086.

[65] Peihua Zhang, Chenggang Wu, Mingfan Peng, Kai Zeng, Ding Yu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. Khaos: The impact of inter-procedural code obfuscation on binary diffing techniques. In *CGO*, page 55–67, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701016. doi: 10.1145/3579990.3580007. URL https://doi.org/10.1145/3579990.3580007.

[66] Peihua Zhang, Chenggang Wu, Hanzhi Hu, Lichen Jia, Mingfan Peng, Jiali Xu, Mengyao Xie, Yuanming Lai, Yan Kang, and Zhe Wang. Shining light on the inter-procedural code obfuscation: Keep pace with progress in binary diffing. *ACM Trans. Archit. Code Optim.*, Just Accepted, October 2024. ISSN 1544-3566. doi: 10.1145/3701992. URL https://doi.org/10.1145/3701992. Just Accepted.

[67] Weiwei Zhang, Shengjian Guo, Hongyu Zhang, Yulei Sui, Yinxing Xue, and Yun Xu. Challenging machine learning-based clone detectors via semantic-preserving code transformations. *IEEE Trans. Softw. Eng.*, 49(5):3052–3070, may 2023. ISSN 0098-5589. doi: 10.1109/TSE.2023.3240118. URL https://doi.org/10.1109/TSE.2023.3240118.

[68] Zhiwei Zhang et al. Dynamic binary analysis and instrumentation. In *SEKE*, pages 329–336, United States of America, 2001. Knowledge Systems Institute.

[69] Andrei Rimsa Álvares, José Nelson Amaral, and Fernando Magno Quintão Pereira. Instruction visibility in spec cpu2017. *Journal of Computer Languages*, 66:101062, 2021. ISSN 2590-1184. doi: https://doi.org/10.1016/j.cola.2021.101062. URL https://www.sciencedirect.com/science/article/pii/S2590118421000411.