Scalable program clone search through spectral analysis

Tristan Benoit, Jean-Yves Marion, Sébastien Bardin

Scalable Program Clone Search Through Spectral Analysis

Tristan Benoit
benoit.tristan.info@gmail.com
Université de Lorraine, CNRS, LORIA
Nancy, France

Jean-Yves Marion
jean-yves.marion@loria.fr
Université de Lorraine, CNRS, LORIA
Nancy, France

Sébastien Bardin
sebastien.bardin@cea.fr
CEA LIST, Université Paris-Saclay
Saclay, France

ABSTRACT

We consider the problem of program clone search, i.e. given a target program and a repository of known programs (all in executable format), the goal is to find the program in the repository most similar to the target program – with potential applications in terms of reverse engineering, program clustering, malware lineage and software theft detection. Recent years have witnessed a blooming in code similarity techniques, yet most of them focus on function-level similarity and function clone search, while we are interested in program-level similarity and program clone search. Actually, our study shows that prior similarity approaches are either too slow to handle large program repositories, or not precise enough, or yet not robust against slight variations introduced by compilers, source code versions or light obfuscations. We propose a novel spectral analysis method for program-level similarity and program clone search called Programs Spectral Similarity (*PSS*). In a nutshell, *PSS* one-time spectral feature extraction is tailored for large repositories, making it a perfect fit for program clone search. We have compared the different approaches with extensive benchmarks, showing that *PSS* reaches a sweet spot in terms of precision, speed and robustness.

CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**; *Malware and its mitigation.*

KEYWORDS

binary code analysis, clone search, spectral analysis

1 INTRODUCTION

Binary code similarity approaches identify similarities or differences [31] between pieces of assembly code (e.g., basic blocks, binary functions or whole programs). We focus on program-level similarities (coined *program similarity* in the following), that is, computing a similarity index between whole programs which is capable of telling at which degree two programs are similar – with potential applications in terms of reverse engineering, program clustering, malware lineage and software theft detection.

Program clone search. Given a query composed of a *target program* and a repository, the *program clone search* ranks repository programs by their program similarity to the target program. The search is successful if the most similar program is a *clone* of the target program. These clones may be (i) compiled with slightly different compiler chains, or (ii) produced from a slightly different version of the source code, or (iii) altered by slight obfuscations.

Applications. Searching program clones between x86 or ARM binaries over a large program repository is necessary when the original program written in source code is unavailable, which happens with commercial off-the-shelf (COTS), legacy programs, firmware or malware. For example, detecting malware clones is a major issue [4, 18, 57, 73], as most malware are actually variants of a few major families active for more than five years[1]. Another application is the identification of libraries [3, 20, 32, 36, 69, 70], which is both a software engineering issue and a cybersecurity issue due to vulnerabilities inside dynamically linked libraries. The problem of library identification, while in between programs and functions in terms of size, is much closer to the case of program clones by its nature, as libraries are not arbitrary collections of functions and require inter-procedural analysis. The situation is similar for patch and firmware analysis [75], or software theft detection [20, 32, 58], which also need to consider a global view of the code.

In all these cases, we see function clone search as only a proxy to a problem that is by nature at the level of programs.

Prior work. Given its potential applications and challenges, the field of similarity detection has been extremely active over the last two decades, starting from the pioneering work of Dullien in 2004 [22, 23] on call-graph isomorphisms and the popular BinDiff tool for recognizing similar binary functions among two related executables. Other approaches include for example symbolic methods [28], graph edit distances [34, 44] and matching techniques [4, 73]. Interestingly, the last five years have seen a strong trend toward machine learning based approaches to binary function similarity [19, 52, 55, 74, 77]. *Overall, most prior work focuses on function clone search and function-level similarity.*

The challenges. Program clone search presents specific challenges compared to standard function similarity. (1) As already stated, it requires comparing programs, i.e. much larger objects than functions, hence similarity checks must be scalable in typical program sizes; (2) We do not consider two programs taken in isolation, but a target program and a (possibly large) program repository, hence the need for very efficient similarity checks that will be iterated over all the programs in the repository; (3) The repository could contain similar but slightly different programs, due to variations in compilers or code versions. Clone search must be robust to such variations; (4) Finally, the technique must work equally well on stripped binary codes (where symbols have been removed at compile time), handle the case where external function names are unavailable (for example IoT device firmware), and handle lightweight obfuscations (such as adding deadcode, or hiding literal identifiers).

All these constraints do not fit well with prior work on similarity, as state-of-the-art is increasingly focused on *function-level* similarities[2], with unclear scalability toward the program-level case. For example, we found in our experiments that SMIT [34] takes more than 43 hours to compute a similarity index between the main library of Geany and the cp command, while DeepBinDiff [21] is

[1] https://www.cisa.gov/uscert/ncas/alerts/aa22-216a
[2] According to Haq and Caballero [31], since 2014, among 40 binary code similarity approaches, only 7 approaches have taken programs as input.

reported to take 10 minutes to compute basic bloc matching on small binaries from the Coreutils package.

Goal. *From the program clone search point of view, there is a strong need for a binary-level program-level similarity technique that is precise, robust to slight variation, and fast enough to operate over large code bases. This is exactly what we want to address in this paper.*

Our proposal. We explore the application of *spectral graph analysis* [14] to the problem of program clone search. It seems a very good starting point as, on graphs, it is both affordable and competitive against graph edit distances (GED) [66] in terms of precision, while GED is arguably a very good (but expensive to compute) notion of graph similarity. Yet, programs are not standard graphs: on the one hand programs seen as graphs can be very large (especially at the binary level), while on the other hand they are highly structured due to their function hierarchy.

We take advantage of this specificity and propose **Program Spectral Similarity (*PSS*)**, the first spectral analysis tailored to *program* similarity. The techniques extract *eigenvalues* related features from both function call graphs and control flow graphs, and take advantage of a *preprocessing* step (done once for the whole program repository) to achieve similarity checks in time *linear in the number of functions of the program* (done for each program in the repository), making it a perfect fit for program clone search – most prior works have at least a quadratic runtime.

We experimentally show that *PSS* outperforms state-of-the-art approaches and is resilient to code variations as well as lightweight obfuscations (e.g., instruction substitution, bogus control flow, control flow flattening). Moreover, *PSS* does not rely on *literal identifiers* (e.g., function names, constant string values), making it robust against a range of basic obfuscations. In our experiments, a program clone search with *PSS* (optimized version) takes on average less than 3s (0.3s and 0.4s for Linux and IoT benchmarks) where, as a comparison, the function embedding Gemini [74] requires roughly 2 minutes per clone search.

We set up a strong comprehensive evaluation framework (14 competitors and 3 baselines) to systematically compare *PSS* with state-of-the-art methods, covering string based methods [69, 70], graph edit distance [27, 34], N-grams [33], vector embedding [19, 52, 55, 74], standard spectral methods [27] and matching algorithms [4, 73]. Our experiments cover our own dataset of diverse open-source projects along with classical Coreutils, Diffutils, Findutils, and Binutils packages along two dimensions (optimization levels and code versions) for a total of 950 programs. Moreover, we consider part of the BinKit dataset [43] (98K samples), covering four optimization levels, 9 compilers, 8 architectures and 4 obfuscations. Finally, we gather 19,959 IoT malware and 84,992 Windows goodware.

Contribution. As a summary, we claim the following:

- A novel technique named *PSS* (together with its optimization PSS_O) for code similarity (Section 4), tailored to *program* clone search over *large* repositories. *PSS* is the first spectral technique tailored to program-level similarity. Especially, *PSS* takes advantage of a preprocessing step to perform latter similarity checks in time linear w.r.t. the number of functions in the program, making it a perfect fit for program clone search over large repositories;
- A comprehensive evaluation framework for program clone search (Section 5), encompassing (1) 97,760 programs from BinKit [43], 19,959 IoT malware, 84,992 Windows programs and a smaller Linux dataset of 950 programs, and (2) three baselines and 14 state-of-the-art methods – 10 of them being reimplemented. *The complete framework is available online*, which is rare in this field [54];
- Experimental evidence (Sections 5) that *PSS* reaches a sweet spot in terms of speed, precision and robustness, making it a perfect fit for program clone search, where prior works in the field are more specialized to function-level similarity evaluation. Especially, *PSS* appears to scale well and to retain good precision in demanding clone search scenarios (cross-compilers, cross-architecture or obfuscation);
- Finally, as another notable result, we show that prior work targeting function clones cannot cope with program clones due to scalability issues.

Besides providing a novel and efficient method for program clone search, our results also shed new light on prior work on code similarity. First, we make the case for the program clone search application scenario and show that it behaves differently enough than the well-studied pairwise function similarity setting, requiring dedicated methods. Second, we are the first to pinpoint the separation in prior work between techniques using literal identifiers and those that do not. As a side result, during our experiments, we identify two simple methods based on literal identifiers (string values and external function names), which despite their simplicity, appear to perform well when these identifiers are available. These methods came from the simplification of ideas coming from the state-of-the-art in library identification by using literal identifiers [20, 32, 69, 73]. Third, we show the potential of dedicated spectral methods for program clone search. Overall, we believe that these results pave the way for novel research directions in the field.

Research artifacts are available on Zenodo [9].

2 PROBLEM STATEMENT

2.1 Program Clone Search Procedure

Given an unknown *target program* P and a *program repository* R, the goal is to identify a *clone* of P in R.

A clone of a program P is defined as follows:

- A program Q compiled from the same source code S as P, but with a different compiler toolchain is a clone of P. For example, P has been compiled with GCC v9.1 using the optimization level -O0 from the source code S, and Q has also been built from S using the same compiler but another optimization level, say -O3;
- A program Q compiled from another version of P source code is a clone of P. For example, both instances of the git application compiled from two source code versions, say v2.35.2 and v2.37.1, are clones.

In the last case, we have to be a bit careful. Indeed, we can only consider incremental versions of an application or library, not major revisions that completely change the source code. In our experiments, the newest and oldest versions of most packages are usually separated by 4 years. However, it goes up to 15 years for the most standard packages: Coreutils, Diffutils, and Findutils.

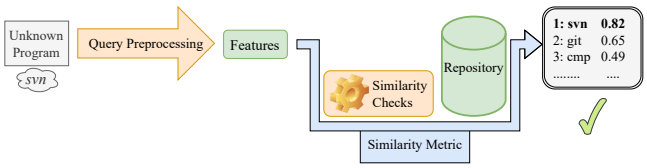

Figure 1: Architecture of a program clone search procedure

Figure 1 illustrates a clone search procedure architecture. Note that all along, we suppose that there is no exact copy of P in the repository R. The repository is a database containing enough information for a clone search procedure. As a result, in practice, a repository is quite an extensive program database w.r.t. the application domain (firmware, plagiarism, malware, etc.).

An evaluation of clone search procedures should take into consideration the three criteria below in order to be realistic:

- The efficiency w.r.t. both the size of the unknown target program and the size of the repository,
- The robustness not only to compiler toolchains but also to slight program variations coming from different source code versions,
- The ability to deal with stripped programs. Moreover, external symbols are not necessarily available when dealing with firmware, lightweight obfuscations, or yet from payload extracted from packers[13].

As we said previously, the main difference between program clone search and function clone search is the size of the binary codes, which is much larger in the case of programs.

At a high level, all program clone search procedures work in a similar way. The repository is already built, and the query process is divided into three steps:

(1) **Query preprocessing.** Upon query, we receive the target program P. We can perform some preprocessing at this step, extracting relevant features for the rest of the procedure;
(2) **Similarity checks.** For each program $Q \in R$, we perform a similarity check with a similarity metric M on (P, Q) – possibly taking advantage of the preprocessing – and record the computed similarity index $M(P, Q)$;
(3) **Decision.** The program Q_{best} with the highest similarity index is considered the most similar. The program clone search succeeds if Q_{best} is a clone of P, otherwise it fails.

2.2 Motivating Example

Let us consider a repository containing 1420 libraries obtained from the compilation of 20 libraries[3] with four optimization levels, five versions of GCC, four versions of clang, and to the 32 and 64 bits x86 platforms. Next, let us imagine we have the 20 libraries as targets (compiled for x86 32 bits with gcc 6.4 and the -O2 optimization level).

[3]From packages libiconv, coreutils, libtool, gss, gdbm, libtasn1, gsl, libmicrohttpd, osip, readline, gsasl, lightning, recutils, gmp, libunistring, and glpk.

Table 1: Clone searches results

Framework	Average precision@1	Total runtime (preprocess. time included)
Asm2Vec [19] †	0.7	35h
Gemini [74] †	1	17h
SAFE [55] †	0.95	160h
αDiff [52] †	1	140h
LibDB [70] †	1	2h
PSS	1	**26s** (includ. 26s of preprocess)

† learning time not included

Lifting function-level clone searches in order to detect program-level clones is attractive. However, to obtain a similarity index between two programs from function embedding methods, we need to find a distance between two sets of function embeddings. Let $embeds(P)$ be the set of function embeddings of a program P. A first solution is to perform a matching between the two sets. Such matching could be an instance of the assignment problem where assigning a function embedding x of P to a function embedding y of P' has a cost $\|x - y\|_2$. However, this problem has complexity $O(n^3)$ where n is the number of functions. We relax the matching so that a function embedding of a program P can be assigned to multiple function embeddings of a program P'.

We define F as the similarity metric for an embedding *embeds*:

$$F(P, P') := - \sum_{x \in embeds(P)} \min_{y \in embeds(P')} \|x - y\|_2 \qquad (1)$$

We consider the following function-level methods and lift them to programs as just explained: *Asm2Vec* [19], *Gemini* [74], *SAFE* [55], *αDiff* [52]. We also consider *LibDB* [70], which is directly designed for libraries (i.e., large pieces of code).

Results. We report in Table 1 the average precision@1, equivalent to the proportion of successful clone searches, as well as clone searches total runtime. *PSS* is precise and successful in all clone searches. Most function-level methods can also find a clone in all clone searches. However, *PSS* takes only 26s in total, while pure function embedding methods take from 17h with *Gemini* to 160h with *SAFE*. Even with pre-filtering, *LibDB* is close to 2h. Moreover, *PSS* runtime is due to its preprocessing; the total similarity checks runtime is negligible. As a result, *PSS* scales up to large repositories with good precision.

3 BACKGROUND

Graph similarity, GED and spectral distance. As programs can be naturally seen as graphs, any good notion of graph similarity is in principle a good candidate for a good program similarity metric.

Graph edit distance (GED) is such a good notion [29]. GED is the smallest cost of an edit path between two graphs, i.e. the smallest transformation going from one of the graphs to the other. Graph edit operations typically include removing or adding a vertex or an edge. Yet, the main drawback of GED is that its computation is NP-hard. Worst, usual approximations have a complexity of $O(n^3)$ [68] where n is the number of nodes in the graph, which is far too expensive for graphs coming from programs. As an example, the graph edit distance method *SMIT* [34] is the slowest method we have tested (cf. Table 4), with 3634 hours of computation on a task where our method takes 1h18m.

The spectral distance between graphs provides an interesting trade-off, as it gives a decent approximation of the graph edit distance between graphs [72] for an affordable linear cost once eigenvalues are computed. We introduce spectral analysis and define spectral distance hereafter.

Spectral (Graph) Analysis. Spectral graph analysis is a method used to investigate properties of graphs by studying the eigenvalues (or, *spectrum*) of standard matrices associated with the graph, such as the adjacency matrix or the Laplacian matrix. Patterns and structures within the graph can be identified, providing key insights about how the graph nodes are interconnected. Distances between graph spectra are called spectral distances. The starting intuition for using graph spectrum is that two isomorphic graphs have the same spectrum; however, the converse is not true. Nevertheless, the spectrum may be used as a proxy for graph similarities.

More formally, an undirected graph $G = (V, E)$ of n vertices is represented by an $n \times n$ adjacency matrix A, where $a_{i,j}$ is one if $(V_i, V_j) \in E$ and zero otherwise. Let d_i be the degree of the vertex V_i. It is useful to compute the Laplacian matrix [14] L of G. An eigenvalue λ and its corresponding eigenvector $\vec{u}$ is a solution to the equation: $(L - \lambda I)\,\vec{u} = \vec{0}$. The spectrum is the set $\{\lambda_1(G), \ldots, \lambda_{|G|}(G)\}$ where $\lambda_1(G) \geq \ldots \geq \lambda_{|G|}(G)$ and where $|G|$ is the number of vertices in G. The enhanced Lanczos algorithm [60] computes the spectrum in time $O(dn^2)$, where d is the average degree of G. We define the spectral distance between G_1 and G_2 (analogous to[39]):

$$sD(G_1, G_2) := \sqrt{\sum_{i=1}^{min(|G_1|,|G_2|)} (\lambda_i(G_1) - \lambda_i(G_2))^2}.$$

4 PROGRAM SPECTRAL SIMILARITY (PSS)

Spectral analysis is suitable for comparing graphs because it provides quantitative metrics, such as spectral distances, which can be used to compare key graph properties regarding connectivity, structure, and distribution. This approach also allows for the normalization of graph size, enabling fair comparisons among varying graph scales. However, computing the spectrum of a graph is cubic in its number of nodes. Therefore, applying spectral analysis to a whole program CFG is too expensive. Moreover, the CFG itself is not stable with respect to compiler toolchains, optimizations and obfuscations.

As a result, our key insight is that a program has more structure than a mere graph: there is a call graph over functions while local functions hold their own control flow graph. We take advantage of this hierarchical structure to devise a quick and stable similarity metric called *Program Spectral Similarity* (*PSS*).

The *PSS* method is based on the combination of two criteria.

- The first measure is the spectral distance between call graphs, including both internal and external calls[4]. Moreover, most compiler optimizations have a small-scale effect on the call graph, as they only impact the content of functions;
- The second measure is a coarse spectral analysis of function control flow graphs, simply considering their number of edges, as it is related to the sum of the eigenvalues as shown below. Since we use only one number to represent a function CFG, we can fit these numbers into a vector comparable to the eigenvalues vectors. Adding function embeddings to the second measure is left for further work.

By the way, we tried to consider only the control flow graphs, and we found that the results were worse than when both above criteria were considered.

The *PSS* method proceeds into two independent steps: the preprocessing step, which is done once and for all, and the similarity check step, which is made for each candidate.

4.1 Preprocessing

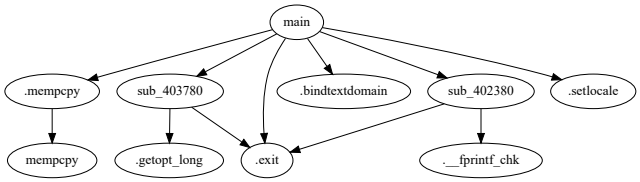

Figure 2: A call graph

Given a program P, the preprocessing first begins by building the function call graph CG of P, including local and external (API) calls. An example of a function call graph is given in Figure 2. It contains external calls such as a call to `mempcpy` as well as local functions such as `sub_403780`. From this, we extract two key vector-features $(\vec{v}, \vec{w})$ of P as follows:

- From an undirected version of the call graph CG, we compute the spectrum $\Lambda = \{\lambda_1(CG), \ldots, \lambda_n(CG)\}$, and we compute $\vec{v} := \frac{\Lambda}{\|\Lambda\|_2}$, the normalized spectrum of the call graph;
- We compute the number of edges $E = (e_1, e_2, \ldots, e_k)$ from each control flow graph F_i of local functions in descending order, and we normalize E as previously: $\vec{w} := \frac{E}{\|E\|_2}$. External functions are ignored at this step since we do not have access to their control flow. Note also that the number of edges is a simple sort of spectral measure since it is related to the spectrum by the relation $2 \times e_i = \sum \lambda_j(F_i)$.

Recall that $||\cdot||_2$ is the Euclidean norm. We normalize features $\vec{v}$ and $\vec{w}$ to deal with differences between program sizes.

[4]Call graphs are useful for a number of tasks. For example, GraphEvo [71] has been able to understand software evolution through call graphs.

4.2 Similarity Check

Given two programs, P_0 and P_1, the preprocessing step has computed features $(\vec{v_0}, \vec{w_0})$ from P_0, and $(\vec{v_1}, \vec{w_1})$ from P_1. The similarity check outputs a similarity index by averaging two measures. The first measure (2) is related to call graphs, while the second (3) is related to function control flow graphs. Then, the similarity metric *PSS* is defined as the average of both above measures (Equation 4).

$$simCG(P_0, P_1) := \sqrt{2} - \sqrt{\sum_{i=0}^{min(|\vec{v_0}|,|\vec{v_1}|)} \left(v_{0,i} - v_{1,i}\right)^2} \quad (2)$$

$$simCFG(P_0, P_1) := \sqrt{2} - \sqrt{\sum_{i=0}^{min(|\vec{w_0}|,|\vec{w_1}|)} \left(w_{0,i} - w_{1,i}\right)^2} \quad (3)$$

$$PSS(P_0, P_1) := \frac{simCG(P_0, P_1) + simCFG(P_0, P_1)}{2\sqrt{2}} \quad (4)$$

4.3 The PSSO Optimization

We found out that *PSS* preprocessing may be quite long over large programs (cf. our own "Windows dataset" in Section 5.5, where computing all eigenvalues of a call graph takes 16.95 seconds per program clone search.). In order to tackle this issue, instead of computing the complete spectrum Λ, we propose to compute only the first K greater eigenvalues so that $\Lambda = \{\lambda_1(CG), \ldots, \lambda_K(CG)\}$. For this, we can take advantage of a variant of the Lanczos algorithm proposed by the ARPACK library [49].

We plot in Figure 3 the preprocessing runtimes and precision scores (see Section 5.2) for different values of K from 30 to 180 on our "Windows data set". We remark that runtimes grow quickly with K, going from 0.06s to 1.31s. On the other hand, there is little change in the precision score between 50 and 150; the score varies from 0.4657 to 0.4664. We select 100 as the value for K since the preprocessing runtime per clone search is only 0.39s, and the precision score is already 0.4661.

We thus propose PSS_O, an optimized version of *PSS* that computes only the first $K = 100$ greater eigenvalues.

4.4 Method Runtimes

Recall that a repository is a database of preprocessed programs. A given unknown target program is first preprocessed, then, from the extracted features, a similarity check is made on the repository. It is clear that the query runtime linearly depends on the size of the repository. In other words, for a repository size of M, and n the number of functions inside a program, if the runtime of a similarity check is $T(n)$ and the preprocessing runtime is $P_T(n)$, then the complexity of a query is bounded by $M \times T(n) + P_T(n)$. As a result, all methods with similarity checks with superlinear time complexity are not feasible over large repositories of large codes, which is confirmed by our experiments.

***PSS* and PSS_O runtimes**. Graphs and Laplacian matrices are sparse in our application domain, offering quick eigenvalues computation. Nevertheless, the complexity of the query prepossessing, described in Section 4.1, is still $O(dn^2)$, where n is the number of functions and d is the average number of calls per function. However, once such prepossessing is done, the runtime of a similarity check, described in Section 4.2, is $O(n)$. Moreover, the runtime of the query preprocessing of PSS_O is reduced to $O(dn)$.

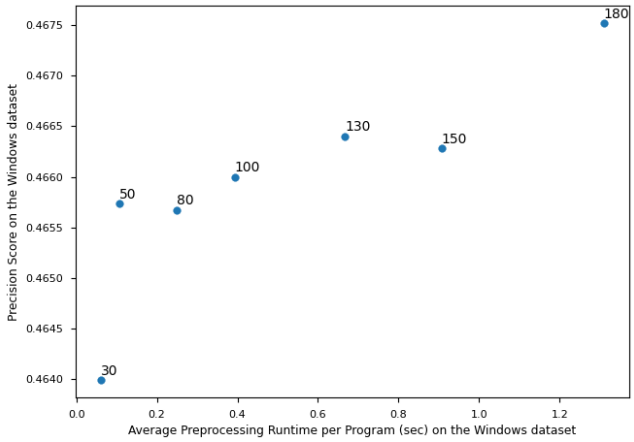

Figure 3: Impact on the Windows dataset of the number of largest eigenvalues computed by PSS optimized version

Table 2: Complexity of program clone search procedures

Method	Class	Similarity check†	Preprocess.‡
SMIT [34]	GED	$O(n^4)$	$O(dn)$
CGC [73]	Matching	$O(n^4)$	$O(dn)$
MutantX-S [33]	**N-gram**	$O(1)$	$O(i)$
Asm2Vec [19]	Functions ML	$O(n^2)$	$O(n)$
Gemini [74]	Functions ML	$O(n^2)$	$O(n)$
SAFE [55]	Functions ML	$O(n^2)$	$O(n)$
αDiff [52]	Functions ML	$O(n^2)$	$O(n)$
LibDX [69]	Strings	$O(s)$	$O(s)$
LibDB [70]	Functions ML and Strings	$O(n^2 + s)$	$O(n + s)$
DeepBinDiff [21]	ML	$O(n^3m^3)$	no preproc.
PSS	**Spectral**	$O(n)$	$O(dn^2)$
PSS$_O$	**Spectral**	$O(n)$	$O(dn)$

n: # functions, i: # instructions , s: # constant string values
d: # calls per function, m: # basic blocks in a function,
† between two programs
‡ performed once for the whole clone search

Comparison with prior work. That is in contrast with function embedding methods which have a similarity check runtime of $O(n^2)$ on this problem using a direct adaptation (see Section 2.2 for further details). Moreover, *DeepBinDiff* [21] contains a step with a linear assignment between basic blocs with a runtime of $O(n^3m^3)$. Worse, both the graph edit distance approximation *SMIT* [34] and the matching method of Xu et al. [73] have a complexity of $O(n^4)$. However, the runtime of *MutantX-S* [33], designed to scale up to large repositories, is only $O(1)$ – yet experiments (Tables 8 and 9 in Section 5.7) show that its robustness is not fully satisfactory.

5 SYSTEMATIC EVALUATION

We evaluate the potential of *PSS* in terms of speed, precision and robustness – the ability to overcome changes in compilation.

Then, we consider here the following Research Questions:

RQ1 What are the fastest methods for clone search?
RQ2 What are the most precise methods for clone search?
RQ3 What are the most robust methods for clone search?
RQ4 What is the impact of each component of PSS?

5.1 Datasets

Basic dataset. We first collect a limited dataset of 950 programs to study the full range of methods along different optimization levels and code versions. The average program has a size of 442 KB. This dataset covers the Coreutils, Diffutils, and Findutils packages compiled with GCC v5.4 on the x86 architecture, and taken from the DeepBinDiff [21] dataset. Moreover, we add the Binutils package as well as 15 open-source projects , including Bash, Code::Blocks, Dia, Graphviz, Geany, Git, Lua, Make, OpenSSH, OpenSSL, Perl, Ruby, SDL, SVN, and VLC, compiled by GCC v9.4 on an x86 architecture. Each unique source code comes in four different *version levels*, and four different *optimization levels*. These programs are all clones of each other.

BinKit dataset. To study scalable methods along different optimization levels, compilers, architectures, and obfuscations, we reuse two Linux programs datasets from BinKit [43]:

- **Normal:** From 51 GNU software packages, 235 unique source codes were extracted. They are compiled with 288 different toolchains for a total of 67,680 programs of an average size of 201 KB. It covers eight architectures (arm, x86, mips, and mipseb, each available in 32 and 64 bits), nine compilers (five versions of GCC and four versions of Clang), and the four optimization levels from -O0 to -O3;
- **Obfuscation:** Four obfuscation options (instruction substitution (SUB), bogus control flow (BCF), control flow flattening (FLA), and all combined) are considered using Obfuscator-LLVM [40] as a compiler. The same architectures and optimization levels as before are covered, for a total of 30,080 programs of an average size of 514 KB.

IoT Malware dataset. We consider 19,959 IoT malware samples, with an average size of 84 KB, from MalwareBazaar[5], submitted between March 2020 and May 2022, spanning 8 architectures (mostly arm, mips, motorola and sparc). Using available meta-data from antivirus reports and YARA rules, we split the data into only three families of clones: 12,357 Mirai, 5,842 Gafgyt, and 1,760 Tsunami.

Windows dataset. We assemble a dataset of 84,992 benign programs running under Windows operating systems (x86, Visual Studio). This amounts to more than 50 GB of raw programs, with an average size of 771 KB. Excluding security updates, the dataset contains more than 28,000 dynamic-link libraries. Samples are divided by target platforms (e.g., Windows 7). We consider that two programs sharing the same file name and the same target platform are clones, yielding 49,443 programs with a clone.

[5]https://bazaar.abuse.ch

5.2 Methodology

A *test field* (T, R) comprises targets set T and a repository R. We break down Basic datasets along version levels and optimization levels. For instance, the test field (-O0,-O1) of the subdataset "Coreutils Option" consists of a repository of Coreutils programs compiled with -O1 paired with the same programs but compiled with -O0 as targets. Similarly, we break down BinKit datasets along optimization levels, compilers, architectures, and obfuscations.

Measures of success: precision@1. Program clone search is an *information retrieval* task. The standard evaluation metrics of information retrieval are precision and recall. This study uses the evaluation metric described in the Asm2Vec paper [19], that is Precision at Position 1 (precision@1). Precision@1 is equal to one if and only if a clone of the target is the most similar program in the repository, as ranked by a similarity metric. We define the precision *score* of a similarity metric as the average precision@1 for every target in every test field against a repository.

5.3 Competitors

We evaluate 14 competitors, 3 baselines and two new heuristics based on literal identifiers (constant string values and external function names) (cf. Table 3). 8 of these frameworks have been adapted **(A)** to the case of program clone search, as it was not their primary objective (e.g., function embedding). Moreover, 10 had to be reimplemented **(R)** because the original implementation was unavailable or due to inherent challenges in effectively utilizing the original implementation within the specific domain of clone search. As highlighted by Marceli et al. [54], code similarity artifacts are rarely available, and even when they are, they are often incomplete.

Baseline. We first investigate basic heuristics such as B_{size}, the size of the program, and D_{size}, the size of the disassembled program. For instance, the similarity metric B_{size} is defined as $B_{size}(a, b) := -|a - b|$, where a and b are program sizes in bits. We also consider a crude shape of the call graph. Let n_1 and e_1 (respectively n_2 and e_2) be the number of vertices and edges of the first (respectively second) call graph. Then the similarity measure *Shape* is defined as:

$$Shape(n_1, e_1, n_2, e_2) := \frac{\min(n_1, n_2)}{\max(n_1, n_2)} \times \frac{\min(m_1, m_2)}{\max(m_1, m_2)}$$

Standard spectral methods. From the spectral method developed by Fyrbiak et al. [27], we derive two methods. The first, *ASCG* **(A) (R)**, is based on the call graph. Let X and Y be the two spectrums in descending order of Laplacians of the two call graphs. There is a normalization $X' := X/X_0$ and $Y' := Y/Y_0$. Then:

$$ASCG(X', Y') := -\sum_{i=0}^{min(|X'|,|Y'|)} \left|X'_i - Y'_i\right|$$

Likewise, we derive a method based on the control flow graph, *ASCFG* **(A) (R)**. Instead of computing the spectrum from the call graph, we select the top 1000 eigenvalues from a reduced control flow graph as vectors X and Y.

Graph edit distance. We implement various basic GED based methods. First, we implement *GED-0* **(A) (R)**, a basic computation of the GED applied between call graphs. The algorithm goes back to the work of Sanfeliu and Fu [66]. Second, we implement *GED-L* **(A) (R)**, a computation of the GED between call graphs with labels.

The algorithm is presented by Fyrbiak et al. [27]. In our application, labels are sets of external function names. Third, we implement the specific GED computation of Hu et al. [34] called *SMIT* **(R)**. We do not integrate the indexing tree of SMIT as we are more interested in their GED measure.

Matchings. We compare with the matching algorithm *CGC* **(R)** from Xu et al. [73]. This algorithm needs three parameters along with a complete classification of mnemonics. We perform preliminary works to find good values for these parameters.

Table 3: Methods included in the evaluation

Framework	Class	A	R	Similarity check	LIR
B_{size}	***Baseline***			$O(1)$	
D_{size}	***Baseline***			$O(1)$	
Shape	***Baseline***			$O(1)$	
ASCG [27]	***Spectral***	×	×	$O(n)$	
ASCFG [27]	***Spectral***	×	×	$O(1)$	
GED-0 [66]	GED	×	×	$O(n^3)$	
MutantX-S [33]	**N-gram**		×	$O(1)$	
Asm2Vec [19]	Function ML	×		$O(n^2)$	
Gemini [74]	Function ML	×		$O(n^2)$	
SAFE [55]	Function ML	×		$O(n^2)$	
DeepBinDiff [21]	ML			$O(n^3m^3)$	
PSS	***Spectral***			$O(n)$	
PSS$_O$	***Spectral***			$O(n)$	
GED-L [27]	GED	×	×	$O(n^3)$	×
SMIT [34]	GED		×	$O(n^4)$	×
CGC [73]	Matching		×	$O(n^4)$	×
αDiff [52]	Function ML	×	×	$O(n^2)$	×
LibDX [69]	Strings		×	$O(s)$	×
LibDB [70]	Strings and Function ML		×	$O(n^2 + s)$	×
StringSet	Strings			$O(s)$	×
FunctionSet	***Strings***			$O(n)$	×

A: Adapted for program clone search, R: Reimplemented
LIR: Some literal identifiers are required

N-gram. We reproduce *MutantX-S* **(R)** from the work of Hu et al. [33]. We extended it to multiple architectures. Each program is represented by the frequencies of 4-grams obtained from the opcode sequence. These frequencies are embedded into a 4096-dimension vector by hashing.

Function embeddings. As previously, we use the similarity metric F to compare sets of vector embeddings (refer to Equation 1 in Section 2.2). We first consider *Asm2Vec* **(A)** [19]. We employ an unsupervised training strategy on the Basic dataset inspired by the original paper. Multiple training phases are performed, with each time one optimization level for training and one for testing. Then, we take *Gemini* **(A)** embedding from Xu et al. [74] in an optimistic setting. We build a version of the basic dataset retaining function names and employ these as ground truths for training. Moreover, we use the embedding of Massarelli et al. [55] with *SAFE* **(A)**. We downloaded a pre-trained model made available by one of the authors[6]. Lastly, we reproduce *αDiff* **(A) (R)** from the framework of Liu et al. [52]. It is tailored to binary function similarity between versions. We sample 25% of the αDiff dataset[7] as our training set. *αDiff* incorporates external function names and in-out degrees in the call graphs.

DeepBinDiff. The framework *DeepBinDiff* from Duan et al. [21] attempts to match basic blocs between two binaries. The similarity metric computes the number of matched basic blocs by *DeepBinDiff* between two programs. Due to its runtime, we were unable to perform experiments, and it is only considered inside the preliminary evaluation.

LibDX. We reproduce the framework *LibDX* **(R)** from Kim et al. [69]. It extracts constant string values from well-defined read-only sections of programs. Constant string values are compared with matchings and the tf–idf statistic.

LibDB. We reproduce the framework *LibDB* **(R)** from Kim et al. [70]. They combine function embeddings and matchings, while using constant string values as pre-filters. We reimplemented *LibDB* with our trained *Gemini* model and ScaNN [30] as the nearest vector search engine.

Function set method. Xu et al. [73] describe a simple method that first matches functions between two programs by using only external function names and mnemonics similarities. Then, the similarity measure is computed by a distance over the two function sets. We simplify this idea and invent the similarity metric *FunctionSet*, which computes the Jaccard similarity index[8] between external function names. Let F_a be the external function names set of a program a. The similarity metric is: $FunctionSet(a, b) := \frac{|F_a \cap F_b|}{|F_a \cup F_b|}$.

String set method . We invent a straightforward metric that compares constant string values inside programs. Let S_a be the set of all constant string values of a program a. The similarity metric is: $StringSet(a, b) := \frac{|S_a \cap S_b|}{|S_a \cup S_b|}$.

We present in Table 3 the characteristics of the different methods considered here. We record the runtime complexity of a similarity check between two programs. We note with n, m, and s, the number of functions, basic blocs in a function CFG, and literal identifiers respectively. We indicate whether a method requires literal identifiers. Note that machine learning approaches require a learning phase, and *Gemini* and *GCG* require manual mnemonics classification.

Implementation. Disassembly is implemented by running the IDA Pro disassembler v7.5 along with a script from the Kam1n0 assembly analysis platform[9]. See the recent survey of Pang et al. [62] on disassembling for more details. Our experiments are run on a cloud server node containing two CPUs with a frequency of 2.10 GHz and 20 cores per CPU. All reported runtimes are equivalent to runtimes using only one core.

[6] https://github.com/facebookresearch/SAFEtorch
[7] https://twelveand0.github.io/AlphaDiff-ASE2018-Appendix
[8] https://en.wikipedia.org/wiki/Jaccard_index
[9] https://github.com/McGill-DMaS/Kam1n0-Community

Table 4: (RQ0) Total runtimes on the Basic dataset

B_{size}	≤ 1h30m
D_{size}	≤ 1h30m
Shape	≤ 1h30m
ASCG	≤ 1h30m
MutantX-S	≤ 1h30m
PSS	≤ 1h30m
PSS_O	≤ 1h30m
LibDX	≤ 1h30m
StringSet	≤ 1h30m
FunctionSet	≤ 1h30m

ASCFG	128h
GED-0	81h
GED-L	46h
SMIT	3634h
CGC	171h
Asm2vec ‡	141h
Gemini ‡	102h
SAFE ‡	655h
αDiff ‡	642h
LibDB ‡	16h

fast methods selected for further analysis
‡: Learning time not included

5.4 Preliminary Evaluation: Method Selection

First, we want to identify methods unable to scale to large benchmarks, in order to not consider them in further analysis. We perform a speed assessment on the basic dataset of 950 programs, and remove the methods unable to achieve it in less than 1h30m.

Results. Results are presented in Table 4. Please note that we could not experiment on DeepBinDiff [21] (with an observed average of more than 10 minutes per similarity check, we estimate that it would have taken more than 20,000h to apply it to the whole basic benchmark), and the training time of ML based methods is not counted in the reported timing. Results show a significant dichotomy between methods, 10 of them being able to succeed in less than 1h30m (often far less), while the other ten methods require far more time (from 16h to 3634h).

Conclusion. This preliminary experiment shows that function-level clone search methods (typically based on ML) [19, 52, 55, 70, 74] or graph-edit distance approaches [27, 34, 66] cannot scale to program-level clone search. In the following, we will consider only the scalable-enough methods, namely our three baselines (B_{size}, D_{size}, *Shape*), as well as *ASCG* [27], *MutantX-S* [33], *LibDX* [70] and our own *PSS*, PSS_O, *StringSet* and *FunctionSet*.

5.5 RQ1: Evaluation of Speed

We report in Table 5 the runtimes and the preprocessing time on each dataset to be fully comprehensive.

Basic. On the Basic dataset containing 950 programs, our method is the slowest and takes 1h18m. Nearly everything is spent during the prepossessing. The adapted spectral method for call graph *ASCG* has similar runtimes. PSS_O takes only 15m8s, the optimization dividing the runtime of *PSS* by more than 5. *LibDX* takes 1m4s, and *StringSet* 38s. The N-gram method *MutantX-S* and the *FunctionSet* method are very fast and take less than ten seconds.

BinKit. On the BinKit dataset, which contains 97,760 programs of an average size of 313 Ko, *PSS* takes 190h, PSS_O 116h, and *MutantX-S* is slower with 220h. With literal identifiers, *LibDX* and *StringSet* are much slower (1965h and 542h, resp.). *FunctionSet* is fast (37h).

IoT Malware. On the IoT dataset containing 19,959 IoT malware, *PSS* takes only 2h9m. It is faster than *MutantX-S* (3h34m). Surprisingly, PSS_O is a bit slower than *PSS* and takes 2h12m. Among

Table 5: (RQ1) Total runtimes. Include preprocessing time. Significant preprocessing times reported in "( )".

Dataset # Programs	Basic 1K	BinKit 98K	IoT 20K	Windows 85K
B_{size}	6s	43h	47m	8h41m
D_{size}	5s	43h	47m	8h45m
Shape	1m22s	21h25m	21m26s	4h16m
ASCG	1h18m	143h	1h23m	243h
preproc.	(1h18m)	(81h)	(19m12s)	(228h)
MutantX-S	4s	220h	3h34m	41h
PSS	1h18m	190h	2h9m	263h
preproc.	(1h18m)	(81h)	(16m42s)	(233h)
PSS_O	15m8s	116h	2h12m	31h29m
preproc.	(15m6s)	(14h3m)	(33m3s)	(5h23m)
LibDX	1m4s	1965h	7h47m	170h
StringSet	38s	542h	9h21m	253h
FunctionSet	3s	37h	7m47s	27h34m

Table 6: (RQ1) Runtimes per clone search (sec). Include preprocess. time. Significant preprocess. times reported in "( )".

Dataset # Programs	Basic 1K	BinKit 98K	IoT 20K	Windows 85K
B_{size}	< 0.01	0.11	0.14	0.63
D_{size}	< 0.01	0.11	0.14	0.63
Shape	0.02	0.05	0.06	0.31
ASCG	1.42 (1.42)	0.37 (0.21)	0.25 (0.06)	17.68 (16.60)
MutantX-S	< 0.01	0.57	0.64	3.00
PSS	1.41 (1.41)	0.49 (0.21)	0.39 (0.05)	19.17 (16.95)
PSS_O	0.27 (0.27)	0.30 (0.04)	0.40 (0.10)	2.29 (0.39)
LibDX	0.02	5.09	1.40	12.43
StringSet	0.01	1.40	1.69	18.47
FunctionSet	< 0.01	0.10	0.02	2.01

methods using literal identifiers, *FunctionSet* is fast, with less than 8 minutes in total. *LibDX* takes 7h47m, while *StringSet* is the slowest with 9h21m.

Windows. *PSS* takes 263h on the Windows dataset. That is far higher than *MutantX-S* (41h) and a bit higher than *StringSet* (253*h*h). However, PSS_O takes less than 32 hours. Table 6 reports average runtimes per clone search. We can see that *PSS* preprocessing time can sometimes be important, e.g., on large Windows binaries (16.95s on similarity checks). First, note that preprocessing time does not increase with the repository size. Second, PSS_O is especially optimized for such cases, and its preprocessing time remains low in all cases.

Conclusion (RQ1)

PSS is often roughly as fast as *MutantX-S* on larger datasets, yet it struggles on large Windows programs. PSS_O remedies this default and is consistently faster than other approaches, but the baselines and *FunctionSet*. Interestingly, *StringSet* is slow on large benchmarks.

Table 7: (RQ2) Precision scores

Dataset	Basic	BinKit	IoT	Windows
B_{size}	0.17	0.166	0.819	0.196
D_{size}	0.16	0.062	0.787	0.445
Shape	0.19	0.297	0.818	0.389
ASCG	0.24	0.554	0.759	0.444
MutantX-S	0.38	0.354	0.870	0.472
PSS	0.38	0.619	0.863	0.475
PSS_O	0.38	0.619	0.862	0.466
LibDX	0.70	0.882	0.707	0.044
StringSet	0.94	0.970	0.922	0.501
FunctionSet	0.87	0.500	0.644	0.426
Random	0.02	0.004	0.477	< 0.001

5.6 RQ2: Evaluation of Precision

We compute precision scores on each dataset. We report the results in Table 7.

BinKit. *PSS* and PSS_O attain a score of 0.619 on BinKit, while the other spectral method *ASCG* has only 0.554. *MutantX-S* is well behind with 0.354. In fact, we show in Table 8 that it achieves scores of 0.01 in cross-architecture scenarios as well as against obfuscations. With literal identifiers, *StringSet* attains 0.970 and *LibDX* 0.882. The *FunctionSet* method has only a score of 0.500.

IoT Malware. *PSS* has a score of 0.863, close to *MutantX-S* (0.870). PSS_O is very close with 0.862, while *ASCG* attains 0.759. With literal identifiers, *StringSet* achieves a score of 0.922. Other literal identifier methods have some troubles. *FunctionSet* has a score of 0.644 because only very few external names are available. Moreover, *LibDX* attains 0.707 because *LibDX* extracts constant string values from read-only sections, which are scarce inside IoT firmware.

Windows. *PSS* attains a score of 0.475 on Windows, just above *MutantX-S* (0.472) and well above *ASCG* (0.444). PSS_O is a bit behind *PSS* and *MutantX-S* with 0.466. Among methods with literal identifiers, *StringSet* attains 0.501. *LibDX* attains only 0.044. Again, *LibDX* extracts constant string values from well-defined read-only sections, which are not prevalent in Windows programs. As before, *FunctionSet* has a rather low score here of only 0.426.

Conclusion (RQ2)

PSS and PSS_O are usually as precise as *MutantX-S* except in cross-architecture and obfuscations scenarios, for which *MutantX-S* fails. When literal identifiers are meaningful, *StringSet* is the most precise method in all datasets, while *FunctionSet* and *LibDX* struggle on IoT and Windows datasets.

5.7 RQ3: Evaluation of Robustness

The last evaluation measures the robustness of the ten clone search methods that survived the speedtest. For this, we consider four scenarios with (i) cross-optimization, (ii) cross-compiler, (iii) cross-architecture and (iv) in the presence of obfuscations. The evaluation leans on the BinKit dataset that we presented earlier.

Results. We report the most crucial test field scores in Table 8. When literal identifiers are available, *StringSet* and *LibDX* are very stable in all scenarios. *FunctionSet* is stable except in scenarios involving cross-architecture because external function names differ between architectures. Note that a strong limitation to this finding is that the considered obfuscations do not hide nor encrypt literal strings and external calls (API), while it is common practice.

PSS and PSS_O are much more robust than *MutantX-S* in cross-architecture, cross-optimization and obfuscations scenarios. For instance, *MutantX-S* falls to 0.02 from the arm to mips architecture, while *PSS* maintains a score of 0.39. The more basic spectral method *ASCG* also falls to 0.08 in this scenario. Interestingly, *PSS* and PSS_O perform better in the cross-architecture test fields than in the (-O0, -O3) and (-O0, -O2) test fields. We hypothesize that while the architecture does not impact that much the produced call graph, advanced optimizations do – function inlining is precisely turned on by the -O2 optimization level in both GCC and Clang.

Statistical analysis. A common pitfall of similarity detection is that a method could in the end consider as similar two programs based on some side aspects (e.g., architecture, compiler used or optimization version) irrelevant from the clone search point of view. We evaluate the sensitivity of the different approaches to such bias by computing rank-biserial correlations between (a) similarity rank in new clone searches and (b) sharing an optimization level. We report average correlations in Table 9 (the lower, the better and less sensitive). *PSS*, PSS_O, *ASCG* and *LibDX* have very small correlations of less than 0.10. On the other hand, the *StringSet* method correlation is moderate (0.45), indicating some bias. Surprisingly, this bias does not seem to impact the robustness of *StringSet* (Table 8). The N-gram method *MutantX-S* has a lower correlation of 0.33 and *FunctionSet* has a small correlation of 0.20.

Conclusion (RQ3)

PSS and PSS_O are robust to cross-optimization, cross-compiler, cross-architecture and obfuscations scenarios, while *MutantX-S* suffers significant precision loss in the cross-architecture and obfuscations cases.

5.8 RQ4: Ablation Study

In Table 10, we report the precision scores of the two components of PSS: *simCG* and *simCFG*. The first is a comparison between eigenvalues of the call graph, while the second is a comparison between the number of edges of functions control flow graphs. *PSS* always attains a higher precision score than *simCG* and *simCFG* on every dataset. We remark that *simCFG* alone is not precise on the Windows dataset (0.163 vs. 0.459 for *simCG*). In Table 11, we report each component's average runtimes per clone search. As expected, *PSS* runtimes are the addition of *simCG* and *simCFG* runtimes. Therefore, *PSS* is at worse one second slower than *simCG*.

Conclusion (RQ4)

PSS is more precise than its components for the price of a slight increase in runtimes.

Table 8: (RQ2,RQ3) Precision scores on the BinKit dataset

Category	Optimization level						Cross-compiler			Cross-architecture				vs. Obfuscation†			
vs.	O0 O1	O0 O2	O0 O3	O1 O2	O1 O3	O2 O3	gcc-4 gcc-8	clang-4 clang-7	clang gcc	arm mips	arm x86	mips x86	32 64	bcf	fla	sub	all
B_{size}	0.04	0.04	0.07	0.19	0.11	0.21	0.11	0.45	0.07	0.03	0.10	0.04	0.04	0.04	0.01	0.08	0.01
D_{size}	0.03	0.03	0.03	0.06	0.05	0.07	0.07	0.09	0.04	0.02	0.05	0.03	0.04	0.02	0.01	0.05	0.01
Shape	0.19	0.07	0.06	0.17	0.11	0.33	0.38	0.65	0.16	0.04	0.16	0.04	0.19	0.25	0.27	0.48	0.23
ASCG	0.40	0.12	0.10	0.43	0.24	0.68	0.78	0.91	0.46	0.08	0.46	0.06	0.59	0.54	0.64	0.78	0.48
MutantX-S	0.04	0.03	0.03	0.43	0.36	0.64	0.67	0.80	0.14	0.02	0.01	0.01	0.06	0.09	0.03	0.54	0.01
PSS	0.54	0.23	0.17	0.59	0.38	0.70	0.79	0.91	0.51	0.39	0.55	0.39	0.66	0.53	0.57	0.82	0.46
PSS_O	0.53	0.24	0.17	0.60	0.39	0.68	0.78	0.90	0.51	0.44	0.54	0.44	0.66	0.52	0.56	0.82	0.46
LibDX	0.89	0.89	0.89	0.89	0.89	0.89	0.89	0.86	0.78	0.87	0.89	0.90	0.88	0.87	0.86	0.86	0.86
StringSet	0.97	0.97	0.97	0.97	0.97	0.97	0.97	0.97	0.97	0.96	0.98	0.96	0.97	0.96	0.97	0.96	0.97
FunctionSet	0.55	0.53	0.53	0.55	0.55	0.56	0.46	0.68	0.55	0.29	0.02	0.00	0.23	0.61	0.61	0.61	0.61

Random clone search results in a precision score inferior to 0.005 on all test fields.
†: The BinKit dataset does not consider any obfuscation of literal identifiers

Table 9: (RQ3) Average rank-biserial correlation for H

Framework	Basic dataset
B_{size}	**0.02**
D_{size}	**0.01**
Shape	**0.04**
ASCG	**0.08**
MutantX-S	0.33

Framework	Basic dataset
PSS	**0.06**
PSS_O	**0.06**
LibDX	**-0.07**
StringSet	0.45
FunctionSet	0.20

Table 10: (RQ4) Components precision scores

Dataset	Basic	BinKit	IoT	Windows
simCG	0.29	0.596	0.856	0.459
simCFG	0.29	0.424	0.856	0.163
PSS	0.38	0.619	0.863	0.475

Table 11: (RQ4) Components runtimes per clone search (sec).
Include preprocess. time. Significant preprocess. times reported in "( )".

Dataset	Basic	BinKit	IoT	Windows
simCG	1.41 (1.41)	0.36 (0.21)	0.22 (0.05)	18.07 (16.95)
simCFG	< 0.01	0.14	0.16	1.06
PSS	1.41 (1.41)	0.49 (0.21)	0.39 (0.05)	19.17 (16.95)

Table 12: Informal summarized comparison

Method	speed	precision	robust.	beware
ASCG [27]	+	-	+	
MutantX-S [33]	+	+	--	
PSS/PSS_O	+/++	+	+	
LibDX [69]	-	++	++	str. extraction str. obf.
StringSet	--	+++	++	str. obf.
FunctionSet	+++	-	-	fun. name obf. static linking

5.9 Summary of Our Main Results

Our novel spectral methods *PSS* and PSS_O reach a sweet spot regarding the trade-off between speed, precision and robustness. They do not need any training phase, scale very well to large repositories and are very robust, even in cross-architecture or cross-compiler scenarios and in case of lightweight obfuscation. Therefore, they are the best candidates for intensive program clone search. Also, it is worth mentioning that direct adaptations of graph based spectral methods lack precision compared to *PSS*, and that the optimization PSS_O is necessary over large programs. A summarized informal comparison with other methods is given in Table 12.

This large study also allowed us to highlight that most prior approaches in the field [19, 52, 55, 74], mostly focused on *function-level* similarity, are far too slow for *program* clone search.

6 RELATED WORKS

Binary code similarities are extensively studied. As a testimony, the review of Haq and Caballero [31] reports numerous input and output granularities on which to study similarities.

Pioneering approaches. Dullien in 2004 [22] introduced a graph based program diffing approach that constructs a call graph isomorphism. A follow-up [23] extended it to match basic blocks inside matched functions. These two results are the basis for the popular BinDiff program diffing plugin for the IDA disassembler. BinDiff aims to recognize similar binary functions among two related executables. In 2006, Kruegel et al. [46] presented an approach based on coloring small graphs with fixed size from the control flow graph to identify structural similarities between different worm mutations. In 2008, Gao et al. proposed BinHunt [28] to find differences between two versions of the same program. BinHunt employs symbolic execution with a constraint solver to prove that two basic blocks implement the same functionality.

Program similarity. The few recent works about program-level similarity [57, 75] have already been thoroughly discussed. Still, we can mention a few more approaches. N-gram methods compare instruction sequences [33, 41, 58, 67]. While we could have employed more fine-grained methods than *MutantX-S* [33] – for

example Exposé [58] considers trigrams inside a function matching, it quickly leads to serious scalability issues. Some other works explore similarities based on dynamic executions and input-output observations [2, 38, 51, 56]. Nevertheless, it is hard to thoroughly explore the execution space with dynamic traces – leading to poor precision, and handling large code repositories requires automating the task of detecting the sources of input and output of all programs in the repository, which can be very complicated. Bruschi et al. [11] tackle the problem of detecting some malware inside a program by matching control flow graphs. But, again, this approach suffers from scalability issues (in the size of the programs) and is thus not amenable to the search over large code repositories. The symbolic method by Luo et al. [53] is robust to simple obfuscations as well as simple changes. However, the running time of symbolic execution is a critical issue on large programs, and anti-analysis obfuscation hinders symbolic approaches [6, 61]. We have already studied the matching method CGC [73]. The complex matching by Xu et al. [73] outperforms a baseline based on external function names and mnemonics. However, we propose *StringSet*, a faster, highly precise method comparing sets of constant string values. A few other matching approaches [4, 10, 48] share the same strengths and weaknesses.

Function similarity. The last five years have seen a tremendous increase in the popularity of binary function similarity with machine learning [19, 52, 55, 74, 77]. Yet, as already discussed, these methods lead to poor scalability when applied to a program similarity setting. More expensive methods than function embeddings do exist. Notably, dynamic analysis seeks to build upon the semantics of binary codes instead of their mere structural properties. BinGo [12] analyzes various execution traces with concepts such as pruning. The work of Hu et al. [35] emulates binary functions to create semantic signatures. Pewny et al. [63] propose to translate binary code to an intermediate representation. This representation allows observing inputs and outputs of basic blocs. These frameworks suffer from the already mentioned pitfalls of dynamic execution: the exploration is either imprecise or very slow. Furthermore, it is unclear how to lift these methods to the case of program similarity, as comparing all functions between multiple codes is costly. Built on the idea of intermediate representation, several approaches [17, 47, 64] perform simplification before comparing. In FirmUp [16], the matching between intermediate representations incorporates multiple functions. The formula has to be transformed into an embedding. The larger the code segment it represents, the less precise the embedding is. Finally, other feature selection methods have been investigated: Rendez-vous [42] extracts statistical features at various granularities, while discovRE [24] and Genius [25] extract features such as the number of arithmetic instructions. Gemini [74] leverages static features from Genius into a machine learning framework.

Source code similarity. Computing source similarities can be performed with different structures such as Abstract Syntax Trees [7, 8, 76] or Program Dependency Graphs [8]. It is also possible to normalize instructions and compare code fragments [8, 37]. Matching tokens, fragments and structures is effective because there is no compiler optimization step which would introduce variations. Moreover, critical information such as types are lost by compilation, while data dependencies are harder to retrieve on binary programs.

Graph similarity. A key question in program similarity is how to compare graphs efficiently. New suggestions for graph similarities include novel graph kernels [26, 45, 59] and the use of machine learning to approximate intractable properties such as graph edit distance [5, 50, 65]. Recently, the work of Bay-Ahmed et al. [1] introduced a new graph similarity metric incorporating both spectral information from the Adjacency Matrix and from the Laplacian. Moreover, the work of Crawford et al. [15] proposed spectral analysis as a similarity metric of real-world networks. Furthermore, the study of Fyrbiak et al. [27] reveals that spectral analysis can compete with more energy-intensive approaches such as GED.

Library identification. The pioneering BAT [32] has proposed three methods for library identification, based on strings, compression algorithms and edit distances between bit sequences. They report that edit distance computations are too costly, while strings can be easily obfuscated. OSSPolice [20] has developed similarity measures based on strings. The special structure of Java programs allows the use of properties such as class and package inclusions [3, 36] in order to identify Android libraries.

7 DISCUSSION AND LIMITATIONS

While *PSS* and PSS_O perform well in our experiments, there are still a number of potential corner cases that must be considered. Generally speaking, these methods will suffer on program clones with very different call graphs. Such differences could come for example: (1) from significant source code revisions – it is why we support only incremental changes of an application or library, (2) or from aggressive inter-procedural compiler optimizations, such as function inlining or function sharing – link-time optimizations may be a growing problem here, (3) or from aggressive inter-procedural obfuscation schemes, such as function merging or virtualization.

Also, as the programs we consider mainly come from C/C++ source codes, it would be interesting to evaluate all the considered methods over programs written in emerging programming languages (e.g., Rust, Go) that may contain language-specific function call patterns.

8 CONCLUSION

We consider the problem of searching program clones in large code repositories. While most prior works have been devoted to function clones, the few existing techniques for program similarity suffer either from scalability issues, low precision, or low robustness to code variations. We propose a novel method called Program Spectral Similarity (*PSS*, and especially its optimized version PSS_O) that reaches a sweet spot in terms of speed, precision, and robustness – even in cross-compiler or cross-architecture setups.

ACKNOWLEDGMENTS

This work is partially supported by: a French PIA grant "Lorraine Université d'Excellence" ANR-15-IDEX-04-LUE, an EU Horizon 2020 research and innovation grant No 830927 (Concordia), and an ANR grant under France 2030 "ANR-22-PECY-0007".

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

REFERENCES

[1] Hadj Ahmed Bay Ahmed, Abdel-Ouahab Boudraa, and Delphine Dare-Emzivat. 2019. A Joint Spectral Similarity Measure for Graphs Classification. *Pattern Recognition Letters* (2019). https://doi.org/10.1016/j.patrec.2018.12.014

[2] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. 2011. Graph-based malware detection using dynamic analysis. *Journal in Computer Virology* (2011). https://doi.org/10.1007/s11416-011-0152-x

[3] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and Its Security Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2976749.2978333

[4] Jinrong Bai, Qibin Shi, and Shiguang Mu. 2019. A Malware and Variant Detection Method Using Function Call Graph Isomorphism. *Security and Communication Networks* (2019). https://doi.org/10.1155/2019/1043794

[5] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. 2019. SimGNN: A Neural Network Approach to Fast Graph Similarity Computation. In *Proceedings of the 12th ACM International Conference on Web Search and Data Mining*. https://doi.org/10.1145/3289600.3290967

[6] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code Obfuscation against Symbolic Execution Attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. https://doi.org/10.1145/2991079.2991114

[7] Ira Baxter, Andrew Yahin, Leonardo de Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. *Proc. of International Conference on Software Maintenance* 368-377, 368–377. https://doi.org/10.1109/ICSM.1998.738528

[8] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering* (2007). https://doi.org/10.1109/TSE.2007.70725

[9] Tristan Benoit. 2023. Artifacts - Scalable Program Clone Search through Spectral Analysis. https://doi.org/10.5281/zenodo.8289599

[10] Martial Bourquin, Andy King, and Edward Robbins. 2013. BinSlayer: Accurate Comparison of Binary Executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. https://doi.org/10.1145/2430553.2430557

[11] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting Self-mutating Malware Using Control-Flow Graph Matching. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. https://doi.org/10.1007/11790754_8

[12] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-Architecture Cross-OS Binary Search. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. https://doi.org/10.1145/2950290.2950350

[13] Binlin Cheng, Jiang Ming, Erika A Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean-Yves Marion. 2021. Obfuscation-Resilient Executable Payload Extraction From Packed Malware. In *USENIX Security Symposium*.

[14] Fan Chung. 1997. *Spectral graph theory*. American Mathematical Society.

[15] Brian Crawford, Ralucca Gera, Jeffrey House, Thomas Knuth, and Ryan Miller. 2016. Graph Structure Similarity using Spectral Graph Theory. *International Workshop on Complex Networks and their Applications* (2016). https://doi.org/10.1007/978-3-319-50901-3_17

[16] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *Proceedings of the 23th International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/3296957.3177157

[17] Yaniv David and Eran Yahav. 2014. Tracelet-Based Code Search in Executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/2594291.2594343

[18] Prasad Deshpande and Mark Stamp. 2016. Metamorphic Malware Detection Using Function Call Graph Analysis. *MIS Review* (2016). https://doi.org/10.31979/etd.t9xm-ahsc

[19] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. *IEEE Symposium on Security and Privacy* (2019). https://doi.org/10.1109/SP.2019.00003

[20] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. *ACM SIGSAC Conference on Computer and Communications Security* (2017). https://doi.org/10.1145/3133956.3134048

[21] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. In *27th Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2020.24311

[22] Thomas Dullien. 2004. Structural Comparison of Executable Objects. *Workshop on Detection of Intrusions and Malware & Vulnerability Assessment* (2004).

[23] Thomas Dullien and Rolf Rolles. 2005. Graph-based comparison of Executable Objects. *SSTIC* (2005).

[24] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *NDSS*. https://doi.org/10.14722/NDSS.2016.23185

[25] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-Based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. https://doi.org/10.1145/2976749.2978370

[26] Aasa Feragen, Niklas Kasenburg, Jens Petersen, Marleen de Bruijne, and Karsten Borgwardt. 2013. Scalable kernels for graphs with continuous attributes. *26th International Conference on Neural Information Processing Systems* (2013). https://doi.org/10.5555/2999611.2999636

[27] Marc Fyrbiak, Sebastian Wallat, Sascha Reinhard, Nicolai Bissantz, and Christof Paar. 2020. Graph Similarity and its Applications to Hardware Security. *IEEE Trans. Comput.* (2020). https://doi.org/10.1109/TC.2019.2953752

[28] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Proceedings on International Conference on Information and Communications Security*. https://doi.org/10.1007/978-3-540-88625-9_16

[29] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. 2010. A survey of graph edit distance. *Pattern Analysis and Applications* (2010). https://doi.org/10.1007/s10044-008-0141-y

[30] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *International Conference on Machine Learning*. https://doi.org/10.5555/3524938.3525302

[31] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *ACM Computing Surveys (CSUR)* (2021). https://doi.org/10.1145/3446371

[32] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. Finding Software License Violations through Binary Code Clone Detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. https://doi.org/10.1145/1985441.1985453

[33] Xin Hu, Sandeep Bhatkar, Kent Griffin, and Kang G. Shin. 2013. MutantX-S: Scalable Malware Clustering Based on Static Features. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. https://doi.org/10.5555/2535461.2535485

[34] Xin Hu, Tzi cker Chiueh, and Kang G. Shin. 2009. Large-scale malware indexing using function-call graphs. In *Proceedings of the ACM Conference on Computer and Communications Security*. https://doi.org/10.1145/1653662.1653736

[35] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary Code Clone Detection across Architectures and Compiling Configurations. In *IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. https://doi.org/10.1109/ICPC.2017.22

[36] Jianjun Huang, Bo Xue, Jiasheng Jiang, Wei You, Bin Liang, Jingzheng Wu, and Yanjun Wu. 2022. Scalably Detecting Third-Party Android Libraries With Two-Stage Bloom Filtering. *IEEE Transactions on Software Engineering* (2022), 1–14. https://doi.org/10.1109/TSE.2022.3215628

[37] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *2012 IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2012.13

[38] Jiyong Jang, Maverick Woo, and David Brumley. 2013. Towards Automatic Software Lineage Inference. In *Proceedings of the 22nd USENIX Conference on Security*. https://doi.org/10.5555/2534766.2534774

[39] Irena Jovanović and Zoran Stanić. 2012. Spectral distances of graphs. *Linear Algebra Appl.* (2012). https://doi.org/10.1016/j.laa.2011.08.019

[40] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15*. https://doi.org/10.1109/SPRO.2015.10

[41] Boojoong Kang, Taekeun Kim, Heejun Kwon, Yangseo Choi, and Eul Gyu Im. 2012. Malware Classification Method via Binary Content Comparison. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*. https://doi.org/10.1145/2401603.2401672

[42] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. 2013. Rendezvous: A search engine for binary code. In *10th Working Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1109/MSR.2013.6624046

[43] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2022. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. *IEEE Transactions on Software Engineering* (2022), 1–23. https://doi.org/10.1109/TSE.2022.3187689

[44] Orestis Kostakis, Joris Kinable, Hamed Mahmoudi, and Kimmo Mustonen. 2011. Improved call graph comparison using simulated annealing. In *Proceedings of the ACM Symposium on Applied Computing*. https://doi.org/10.1145/1982185.1982509

[45] Nils M. Kriege, Pierre-Louis Giscard, and Richard C. Wilson. 2016. On Valid Optimal Assignment Kernels and Applications to Graph Classification. In *30th International Conference on Neural Information Processing Systems*. https://doi.org/10.5555/3157096.3157278

[46] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. 2006. Polymorphic worm detection using structural information of

executables. *International Workshop on Recent Advances in Intrusion Detection* (2006). https://doi.org/10.1007/11663812_11

[47] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. 2013. Fast Location of Similar Code Fragments Using Semantic 'Juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. https://doi.org/10.1145/2430553.2430558

[48] Yeo Reum Lee, BooJoong Kang, and Eul Gyu Im. 2013. Function Matching-Based Binary-Level Software Similarity Calculation. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*. https://doi.org/10.1145/2513228.2513300

[49] R. B. Lehoucq, D. C. Sorensen, and C. Yang. 1998. *ARPACK Users' Guide*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719628

[50] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph matching networks for learning the similarity of graph structured objects. In *36th International conference on machine learning*.

[51] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. 2012. Lines of Malicious Code: Insights into the Malicious Software Industry. In *Proceedings of the 28th Annual Computer Security Applications Conference*. https://doi.org/10.1145/2420950.2421001

[52] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αDiff: Cross-Version Binary Code Similarity Detection with DNN. In *33rd IEEE/ACM International Conference on Automated Software Engineering*. https://doi.org/10.1145/3238147.3238199

[53] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. *IEEE Transactions on Software Engineering* (2017). https://doi.org/10.1109/TSE.2017.2655046

[54] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, and Yanick Fratantonio. 2022. How Machine Learning Is Solving the Binary Function Similarity Problem. In *31st USENIX Security Symposium (USENIX Security 22)*.

[55] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2021. Function Representations for Binary Similarity. *IEEE Transactions on Dependable and Secure Computing* (2021). https://doi.org/10.1109/TDSC.2021.3051852

[56] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *USENIX Security Symposium*. https://doi.org/10.5555/3241189.3241211

[57] Jiang Ming, Dongpeng Xu, and Dinghao Wu. 2015. Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference. In *IFIP Advances in Information and Communication Technology*. https://doi.org/10.1007/978-3-319-18467-8_28

[58] Beng Heng Ng and Atul Prakash. 2013. Expose: Discovering Potential Binary Code Re-use. In *37th IEEE Annual Computer Software and Applications Conference*. https://doi.org/10.1109/COMPSAC.2013.83

[59] Giannis Nikolentzos, Polykarpos Meladianos, Stratis Limnios, and Michalis Vazirgiannis. 2018. A Degeneracy Framework for Graph Similarity. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI-18*. https://doi.org/10.5555/3304889.3305021

[60] Irving Ojalvo and Miya Newman. 1970. Vibration modes of large structures by an automatic matrix-reduction method. *Aiaa Journal - AIAA J* (1970). https://doi.org/10.2514/3.5878

[61] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. 2019. How to Kill Symbolic Deobfuscation for Free (or: Unleashing the Potential of Path-Oriented Protections). In *Proceedings of the 35th Annual Computer Security Applications Conference*. https://doi.org/10.1145/3359789.3359812

[62] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP40001.2021.00012

[63] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-Architecture Bug Search in Binary Executables. In *IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2015.49

[64] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging Semantic Signatures for Bug Search in Binary Programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*. https://doi.org/10.1145/2664243.2664269

[65] Pau Riba, Andreas Fischer, Josep Lladós, and Alicia Fornés. 2021. Learning graph edit distance by graph neural networks. *Pattern Recognition* (2021). https://doi.org/10.1016/j.patcog.2021.108132

[66] Alberto Sanfeliu and King-Sun Fu. 1983. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics* (1983). https://doi.org/10.1109/TSMC.1983.6313167

[67] Igor Santos, Felix Brezo, Javier Nieves, Yoseba K. Penya, Borja Sanz, Carlos Laorden, and Pablo G. Bringas. 2010. Idea: Opcode-Sequence-Based Malware Detection. In *Engineering Secure Software and Systems*, Fabio Massacci, Dan Wallach, and Nicola Zannone (Eds.). https://doi.org/10.1007/978-3-642-11747-3_3

[68] Francesc Serratosa. 2014. Fast computation of Bipartite graph matching. *Pattern Recognition Letters* (2014). https://doi.org/10.1016/j.patrec.2014.04.015

[69] W. Tang, P. Luo, J. Fu, and D. Zhang. 2020. LibDX: A Cross-Platform and Accurate System to Detect Third-Party Libraries in Binary Code. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, 104–115. https://doi.org/10.1109/SANER48275.2020.9054845

[70] Wei Tang, Yanlin Wang, Hongyu Zhang, Shi Han, Ping Luo, and Dongmei Zhang. 2022. LibDB: An Effective and Efficient Framework for Detecting Third-Party Libraries in Binaries. In *Proceedings of the 19th International Conference on Mining Software Repositories*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3524842.3528442

[71] Vijay Walunj, Gharib Gharibi, Duy H. Ho, and Yugyung Lee. 2019. GraphEvo: Characterizing and Understanding Software Evolution using Call Graphs. In *2019 IEEE International Conference on Big Data (Big Data)*. https://doi.org/10.1109/BigData47090.2019.9005560

[72] Richard C. Wilson and Ping Zhu. 2008. A study of graph spectra for comparing graphs and trees. *Pattern Recognition* (2008). https://doi.org/10.1016/j.patcog.2008.03.011

[73] Ming Xu, Lingfei Wu, Shuhui Qi, Jian Xu, Haiping Zhang, Yizhi Ren, and Ning Zheng. 2013. A similarity metric method of obfuscated malware using function-call graph. *Journal of Computer Virology and Hacking Techniques* (2013). https://doi.org/10.1007/s11416-012-0175-y

[74] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. https://doi.org/10.1145/3133956.3134018

[75] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: Security Patch Analysis for Binaries towards Understanding the Pain and Pills. In *IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2017.49

[76] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*. https://doi.org/10.1145/2420950.2421003

[77] Jia Yang, Cai Fu, Xiao-Yang Liu, Heng Yin, and Pan Zhou. 2021. Codee: A Tensor Embedding Scheme for Binary Code Search. *IEEE Transactions on Software Engineering* (2021). https://doi.org/10.1109/TSE.2021.3056139