Scalable program clone search through spectral analysis
Tristan Benoit, Jean-Yves Marion, Sébastien Bardin

To cite this version:
Tristan Benoit, Jean-Yves Marion, Sébastien Bardin. Scalable program clone search through spectral analysis. ESEC/FSE ’23 - 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Dec 2023, San Francisco, United States. 10.48550/arXiv.2210.13063 . hal-03826726v4

HAL Id: hal-03826726
https://hal.science/hal-03826726v4
Submitted on 1 Sep 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 years1. 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 similarities2, 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

1https://www.cisa.gov/uscert/ncas/alerts/aa22-216a
2According 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 PSSO) 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.

<table>
<thead>
<tr>
<th>Framework</th>
<th>Average precision@1</th>
<th>Total runtime (preprocess. time included)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asm2Vec [19] †</td>
<td>0.7</td>
<td>35h</td>
</tr>
<tr>
<td>Gemini [74] †</td>
<td>1</td>
<td>17h</td>
</tr>
<tr>
<td>SAFE [55] †</td>
<td>0.95</td>
<td>160h</td>
</tr>
<tr>
<td>αDiff [52] †</td>
<td>1</td>
<td>140h</td>
</tr>
<tr>
<td>LibDB [70] †</td>
<td>1</td>
<td>2h</td>
</tr>
<tr>
<td>PSS</td>
<td>1</td>
<td>26s (includ. 26s of preprocess)</td>
</tr>
</tbody>
</table>

† learning time not included

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.

2.2 Motivating Example

Let us consider a repository containing 1420 libraries obtained from the compilation of 20 libraries9 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).

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:

\[
F(P, P') := \min_{x \in \text{embeds}(P)} \min_{y \in \text{embeds}(P')} \| x - y \|_2
\]

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 $\lambda$ and its corresponding eigenvector $\tilde{u}$ is a solution to the equation: $(L - \lambda I) \tilde{u} = 0$. The spectrum is the set $\{\lambda_1(G), \ldots, \lambda_n(G)\}$ where $\lambda_1(G) \geq \ldots \geq \lambda_n(G)$ 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

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 $(\tilde{v}, \tilde{w})$ 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 $\tilde{v} := \frac{\Lambda}{||\Lambda||_1}$, 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: $\tilde{w} := \frac{E}{||E||_1}$. 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 $\tilde{v}$ and $\tilde{w}$ 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 \((\omega_0, \omega_0')\) from \( P_0 \), and \((\omega_1, \omega_1')\) 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).

\[
\text{simCG}(P_0, P_1) := \sqrt{2} - \sqrt{\sum_{i=0}^{\min(|\omega_0|, |\omega_1|)} (\omega_{0,i} - \omega_{1,i})^2}
\]

\[
\text{simCFG}(P_0, P_1) := \sqrt{2} - \sqrt{\sum_{i=0}^{\min(|\omega_0|, |\omega_1|)} (\omega_{0,i} - \omega_{1,i})^2}
\]

\[
PSS(P_0, P_1) := \frac{\text{simCG}(P_0, P_1) + \text{simCFG}(P_0, P_1)}{2\sqrt{2}}
\]

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 \( \Lambda \), 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 \( \text{PSSO} \), 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.

\( \text{PSS} \) and \( \text{PSSO} \) runtimes. Graphs and Laplacian matrices are sparse in our application domain, offering quick eigenvalues computation. Nevertheless, the complexity of the query preprocessing, 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 \( \text{PSSO} \) is reduced to \( O(dn) \).

Comparison with prior work. That 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 blocks with a runtime of \( O(n^3 m^3) \). Worse, both the graph edit distance approximation \( \text{SMIT} \) [34] and the matching method of Xu et al. [73] have a complexity of \( O(n^4) \). However, the runtime of \( \text{MutantX-S} \) [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. Therefore, 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 MalwareBazaar⁷, 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.

5.2 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 Marcelli 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 Bsize, the size of the program, and Dsize, the size of the disassembled program. For instance, the similarity metric Bsize is defined as $\text{Bsize}(a,b) := |a - b|$, 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:

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

**Standard spectral methods.** From the spectral method developed by Fyriák et al. [26], 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:

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

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 Sanfelio and Fu [66]. Second, we implement GED-L (A) (R), a computation of the GED between call graphs with labels.

---

⁷https://bazaar.abuse.ch
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**

<table>
<thead>
<tr>
<th>Framework</th>
<th>Class</th>
<th>A</th>
<th>R</th>
<th>Similarity check</th>
<th>LIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bsize</td>
<td>Baseline</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>Dsize</td>
<td>Baseline</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>Shape</td>
<td>Baseline</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>ASCG [27]</td>
<td>Spectral</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>ASCFG [27]</td>
<td>Spectral</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>GED-0 [66]</td>
<td>N-gram</td>
<td>O(n)</td>
<td>O(n^4)</td>
<td>O(n^5)</td>
<td></td>
</tr>
<tr>
<td>MutantX-S [33]</td>
<td>Function ML</td>
<td>O(n)</td>
<td>O(n^4)</td>
<td>O(n^5)</td>
<td></td>
</tr>
<tr>
<td>Asm2Vec [19]</td>
<td>Function ML</td>
<td>O(n)</td>
<td>O(n^4)</td>
<td>O(n^5)</td>
<td></td>
</tr>
<tr>
<td>Gemini [74]</td>
<td>Function ML</td>
<td>O(n)</td>
<td>O(n^4)</td>
<td>O(n^5)</td>
<td></td>
</tr>
<tr>
<td>SAFE [55]</td>
<td>Function ML</td>
<td>O(n)</td>
<td>O(n^4)</td>
<td>O(n^5)</td>
<td></td>
</tr>
<tr>
<td>DeepBinDiff [21]</td>
<td>Spectral</td>
<td>O(n)</td>
<td>O(n^4)</td>
<td>O(n^5)</td>
<td></td>
</tr>
</tbody>
</table>

Note that machine learning approaches require a learning phase, respectively. We indicate whether a method requires literal identifiers.

**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. Lastly, we reproduce aDiff (A) (R) from the framework of Liu et al. [52]. It is tailored to binary function similarity between versions. We sample 25% of the aDiff dataset as our training set. aDiff incorporates external function names and out-in 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 between external function names. Let $F_a$ be the external function names set of a program $a$. The similarity metric is: $\text{FunctionSet}(a, b) := \frac{|F_a \cap F_b|}{|F_a \cup F_b|}$.

**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: $\text{StringSet}(a, b) := \frac{|S_a \cap S_b|}{|S_a \cup S_b|}$.

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. 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.

---

1. https://github.com/facebookresearch/SAFEtorch
2. https://twelveand0.github.io/AlphaDiff-ASE2018-Appendix
4. https://github.com/McGill-DMaS/Kam1n0-Community
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\text{size}, D\text{size}, Shape), as well as ASCG [27], MutantX-S [33], LibDX [70] and our own PSS, PSS\text{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 preprocessing. The adapted spectral method for call graph ASCG has similar runtimes. PSS\text{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\text{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\text{O} is a bit slower than PSS and takes 2h12m. Among 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 (253hh). However, PSS\text{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\text{O} is especially optimized for such cases, and its preprocessing time remains low in all cases.

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Basic</th>
<th>BinKit</th>
<th>IoT</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td># Programs</td>
<td>1K</td>
<td>98K</td>
<td>20K</td>
<td>85K</td>
</tr>
<tr>
<td>B\text{size}</td>
<td>6s</td>
<td>43h</td>
<td>47m</td>
<td>8h41m</td>
</tr>
<tr>
<td>D\text{size}</td>
<td>5s</td>
<td>43h</td>
<td>47m</td>
<td>8h45m</td>
</tr>
<tr>
<td>Shape</td>
<td>1m22s</td>
<td>21h25m</td>
<td>21m26s</td>
<td>4h16m</td>
</tr>
<tr>
<td>ASCG</td>
<td>1h18m</td>
<td>143h</td>
<td>1h23m</td>
<td>243h</td>
</tr>
<tr>
<td>Shape</td>
<td>4s</td>
<td>220h</td>
<td>3h34m</td>
<td>41h</td>
</tr>
<tr>
<td>PSS</td>
<td>1h18m</td>
<td>190h</td>
<td>2h9m</td>
<td>263h</td>
</tr>
<tr>
<td>PSS\text{O}</td>
<td>15m8s</td>
<td>116h</td>
<td>2h12m</td>
<td>31h29m</td>
</tr>
<tr>
<td>StringSet</td>
<td>38s</td>
<td>542h</td>
<td>9h21m</td>
<td>253h</td>
</tr>
<tr>
<td>FunctionSet</td>
<td>3s</td>
<td>37h</td>
<td>7m47s</td>
<td>27h34m</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Basic</th>
<th>BinKit</th>
<th>IoT</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td># Programs</td>
<td>1K</td>
<td>98K</td>
<td>20K</td>
<td>85K</td>
</tr>
<tr>
<td>B\text{size}</td>
<td>&lt; 0.01</td>
<td>0.11</td>
<td>0.14</td>
<td>0.63</td>
</tr>
<tr>
<td>D\text{size}</td>
<td>&lt; 0.01</td>
<td>0.11</td>
<td>0.14</td>
<td>0.63</td>
</tr>
<tr>
<td>Shape</td>
<td>0.02</td>
<td>0.05</td>
<td>0.06</td>
<td>0.31</td>
</tr>
<tr>
<td>ASCG</td>
<td>1.42 (1.42)</td>
<td>0.37 (0.21)</td>
<td>0.25 (0.06)</td>
<td>17.68 (16.60)</td>
</tr>
<tr>
<td>MutantX-S</td>
<td>&lt; 0.01</td>
<td>0.57</td>
<td>0.64</td>
<td>3.00</td>
</tr>
<tr>
<td>PSS</td>
<td>1.41 (1.41)</td>
<td>0.49 (0.21)</td>
<td>0.39 (0.05)</td>
<td>19.17 (16.95)</td>
</tr>
<tr>
<td>PSS\text{O}</td>
<td>0.27 (0.27)</td>
<td>0.30 (0.04)</td>
<td>0.40 (0.10)</td>
<td>2.29 (0.39)</td>
</tr>
<tr>
<td>LibDX</td>
<td>0.02</td>
<td>5.09</td>
<td>1.40</td>
<td>12.43</td>
</tr>
<tr>
<td>StringSet</td>
<td>0.01</td>
<td>1.40</td>
<td>1.69</td>
<td>18.47</td>
</tr>
<tr>
<td>FunctionSet</td>
<td>&lt; 0.01</td>
<td>0.10</td>
<td>0.02</td>
<td>2.01</td>
</tr>
</tbody>
</table>

Conclusion (RQ1)

PSS is often roughly as fast as MutantX-S on larger datasets, yet it struggles on large Windows programs. PSS\text{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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Basic</th>
<th>BinKit</th>
<th>IoT</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_size</td>
<td>0.17</td>
<td>0.166</td>
<td>0.819</td>
<td>0.196</td>
</tr>
<tr>
<td>D_size</td>
<td>0.16</td>
<td>0.062</td>
<td>0.787</td>
<td>0.445</td>
</tr>
<tr>
<td>Shape</td>
<td>0.19</td>
<td>0.297</td>
<td>0.818</td>
<td>0.389</td>
</tr>
<tr>
<td>ASCG</td>
<td>0.24</td>
<td>0.554</td>
<td>0.759</td>
<td>0.444</td>
</tr>
<tr>
<td>MutantX-S</td>
<td>0.38</td>
<td>0.354</td>
<td>0.870</td>
<td>0.472</td>
</tr>
<tr>
<td>PSS</td>
<td>0.38</td>
<td>0.619</td>
<td>0.863</td>
<td>0.475</td>
</tr>
<tr>
<td>PSS_O</td>
<td>0.38</td>
<td>0.619</td>
<td>0.862</td>
<td>0.466</td>
</tr>
<tr>
<td>LibDX</td>
<td>0.70</td>
<td>0.882</td>
<td>0.707</td>
<td>0.044</td>
</tr>
<tr>
<td>StringSet</td>
<td>0.94</td>
<td>0.970</td>
<td>0.922</td>
<td>0.501</td>
</tr>
<tr>
<td>FunctionSet</td>
<td>0.87</td>
<td>0.500</td>
<td>0.644</td>
<td>0.426</td>
</tr>
<tr>
<td>Random</td>
<td>0.02</td>
<td>0.004</td>
<td>0.477</td>
<td>&lt; 0.001</td>
</tr>
</tbody>
</table>

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.

**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 (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. It is the first 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.

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

<table>
<thead>
<tr>
<th>Framework</th>
<th>BinKit</th>
<th>IoT</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>simCFG</td>
<td>1.41(1.41)</td>
<td>0.42</td>
<td>0.17</td>
</tr>
<tr>
<td>PSS</td>
<td>1.41(1.41)</td>
<td>0.49</td>
<td>0.39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Basic</th>
<th>BinKit</th>
<th>IoT</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>simCFG</td>
<td>0.29</td>
<td>0.36(0.21)</td>
<td>0.22(0.05)</td>
<td>18.07(16.95)</td>
</tr>
<tr>
<td>PSS</td>
<td>0.38</td>
<td>0.619</td>
<td>0.863</td>
<td>0.475</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Framework</th>
<th>Basic dataset</th>
<th>Framework</th>
<th>Basic dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>DirSize</td>
<td>0.02</td>
<td>LibDX</td>
<td>-0.07</td>
</tr>
<tr>
<td>DSize</td>
<td>0.01</td>
<td>PSS</td>
<td>0.06</td>
</tr>
<tr>
<td>Shape</td>
<td>0.33</td>
<td>FunctionSet</td>
<td>0.45</td>
</tr>
<tr>
<td>ASCG</td>
<td>0.08</td>
<td>StringSet</td>
<td>0.20</td>
</tr>
<tr>
<td>MutantX-S</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

5.9 Summary of Our Main Results

Our novel spectral methods PSS and PSSO 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 PSSO 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, 54, 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 in 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 blocks. 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 PSSO 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 PSSO) 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


[34] Christoph Kruegel, Engin Kirda, Darren Mutz, William Roberts, and Giovanni Vigna. 2006. Polymorphic worm detection using structural information of

Tristan Benteil, Jean-Yves Marion, and Sébastien Bardin
Scalable Program Clone Search Through Spectral Analysis

binary disassembly but were afraid to ask. In 2021 IEEE Symposium on Security and Privacy (SP). https://doi.org/10.1109/SP40001.2021.00012