# Scalable program clone search through spectral analysis

Tristan Benoit, Jean-Yves Marion, Sébastien Bardin

# Scalable Program Clone Search Through Spectral Analysis

Tristan Benoit
*Université de Lorraine, NRS*,
LORIA,
F-54000 Nancy, France
tristan.benoit@loria.fr

Jean-Yves Marion
*Université de Lorraine, CNRS*,
LORIA,
F-54000 Nancy, France
jean-yves.marion@loria.fr

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

***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 our 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 while we are interested in program-level similarity. Consequently, these recent approaches are not directly suited to program clone search, being either too slow to handle large code bases, not precise enough, or not robust against slight variations introduced by compilation or source code versions. We introduce Programs Spectral Similarity (*PSS*), the first spectral analysis dedicated to program-level similarity. *PSS* reaches a sweet spot in terms of precision, speed and robustness. Especially, its one-time spectral feature extraction is tailored for large repositories of programs, making it a perfect fit for program clone search.**

## I. Introduction

Binary code similarity approaches identify similarities or differences [27] 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. Searching similarities 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 and malware. Program similarity has a range of real-world applications such as malware analysis [5], [15], [47], [62] (e.g., lineage, clustering, detection), patch and firmware analysis [64], or software theft detection [48].

Given its potential applications and challenges, this field of research has been extremely active over the last two decades, starting from the pioneering work of Dullien in 2004 [18], [19] on call-graph isomorphism and the popular BinDiff tool for recognizing similar binary functions among two related executables. Other approaches include for example symbolic methods [24], graph-edit distances [29], [35] and matching techniques [5], [62]. Interestingly, the last five years have seen a strong trend toward machine learning based approaches to binary function similarity [16], [42], [45], [63], [65].

**Program clone search**. Given a query composed of a *target program* and a repository, the *program clone search* ranks repository programs by their similarity to the target program. The search is successful if the most similar program is a *clone* of the target program. These clones could have been (i) compiled from the same source code with a different compiler chain, or (ii) produced using a slightly different version of the source code. The case of obfuscation is left as future work.

**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 compiler optimizations 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) and handle the case where external function names are unavailable, for example IoT device firmware.

All these constraints do not fit well with prior work on similarity, as state-of-the-art is increasingly focused on *function-level* similarities [1], with unclear scalability toward the program-level case.

For example, we found in our experiments that SMIT [29] takes more than 43 hours to compute a similarity index between the main library of Geany and the cp command (see Section V-G, Table X), while DeepBinDiff [17] is reported to take 10 minutes to compute basic bloc matching on small binaries from the Coreutils package.

**Problem and 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 be able 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* [12] to the problem of program clone search. It seems a very good starting point as, on graphs, it is both cheap and competitive against graph-edit distances [57] in terms of

[1] According to Haq and Caballero [27], since 2014, among 40 binary code similarity approaches, only 7 approaches have taken programs as input.

precision. 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 are at least quadratic.

We experimentally show that *PSS* outperforms state-of-the-art approaches and is resilient to slight syntactic code variations. In our experiments, a program clone search with *PSS* takes on average 2s, all the time being spent in the preprocessing phase that scales very well on large repositories, while the time for similarity checks is extremely low (close to 0 in our experiments). As a comparison, the function embedding Gemini [63] requires more than 15 minutes per clone search.

We set up a strong comprehensive evaluation framework (13 competitors and 3 baselines) to systematically compare PSS with state-of-the-art methods, covering graph-edit distance [23], [29], N-grams [28], vector embedding [16], [42], [45], [63], standard spectral methods [23] and matching algorithms [5], [62]. Our experiments cover three benchmarks along two dimensions (optimization levels and code versions) and comprise diverse open-source projects along with classical Coreutils, Findutils, Diffutils and Binutils packages for a total of 1,108 programs. We investigate four scenarios, revealing the impact of confounding factors such as optimization level or code version as well as the impact of the number of clones inside a repository. After this comprehensive evaluation, we investigate two case studies on the most promising approaches: (1) a classification of $19,959$ IoT malware, and (2) large repositories (up to $84,992$) of Windows programs. *All this evaluation framework will be publicly made available.*

Finally, this systematic study leads as well to a few notable side results: (i) First, we found out that (what we call) the *function set method* (*FunctionSet*), a mere Jaccard distance over the set of external function names, actually performs very well on medium size repositories of standard Linux binaries where such external names are available (Section V) – PSS ranks still second in that case and beats other methods relying on external names. (ii) Second, our experiments (Section V) show that some state-of-the-art techniques perform worse than basic baselines, based for example on code size. (iii) Third, the case studies on Windows binaries and IoT malware show that in these settings the *function set method* is less efficient than *PSS* (Sections VI-VII). (iv) Fourth, a large benchmark of Windows binaries demonstrates *PSS* ability to scale up to large repositories (Section VII). A clone search on a repository of size $84,992$ takes $15.51$ seconds (2.09s for similarity checks), very close to the $14.53$ seconds necessary on a repository of size $42,648$ (1.04s for similarity checks) – and $13,97$ seconds for size $21,113$ (0.52s for similarity checks).

**Contribution**. As a summary, this paper makes the following contribution:

- A novel technique named PSS for code similarity (Section IV), 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 m evaluation (Section V), encompassing over $1,108$ Linux programs, four scenarios, three baselines and 13 state-of-the-art methods – 9 of them being reimplemented. *The complete framework is available online*, which is rare in this field [44]. Moreover, we add two case studies on the classification of 19,959 IoT malware (Section VI) and on 84,992 Windows programs (Section VII);
- Experimental evidence (Sections V,VI and VII) 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;
- The identification of the simple *FunctionSet* method as the best alternative for program clone search on medium repositories of standard Linux binaries (Section V) when external function names are available. To the best of our knowledge, this is the first time this simple technique is identified in the context of code similarity;
- The highlight that some state-of-the-art approaches seem to be inferior or at best roughly equivalent to very basic baselines (Section V). In recent prior work, only Xu et al. [62] consider comparisons to baselines. There is a call to action here for the community.

Besides providing a novel and efficient method for program clone search, our results shed new light on prior work on code similarity. Primarily, we introduce the program clone search application scenario and show that it behaves differently enough than the well-studied pairwise function similarity setting, requiring dedicated methods. Also, we are the first to pinpoint the separation in prior work between techniques using external function names and those which do not, and identify the simple *FunctionSet* method as competitive when external function names are available. Finally, we show that some prior methods are not better in our setting than basic baselines. These baselines should be systematically considered in further experimental evaluations for code similarity.

We believe that these results pave the way for novel research directions in the field, as well as for revisiting current evaluation practices.

*Our implementations and benchmarks are available at:*

*https://github.com/sppunderreview/PSS.*

## II. Problem statement and motivating example

### A. Program clone search procedure

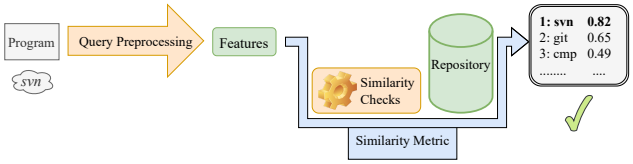



Fig. 1: Architecture of a program clone search procedure.

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 $O_0$ from the source code $S$, and $Q$ has also been built from $S$ using the same compiler but another optimization level, say $O_3$;
- 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.

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 [11].

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 is successful if $Q_{best}$ is a clone of $P$. Otherwise, it is a failure.

### B. Motivating example

Let us consider a repository containing 83 programs from open-source projects including *Bash*, *Coreutils*, *Dia*, *Diffutils*, *Graphviz*, *Geany*, *Git*, *Lua*, *Make*, *OpenSSH*, *OpenSSL*, *Perl*, *Ruby*, *SDL*, *SVN*, and *VLC*. Now, let us imagine our goal is to find a clone of an unknown library, say the *VLC* core library compiled by GCC v9.3 with optimization level $O_0$, inside this repository. If the program search clone procedure returns the correct answer, we will know the library name.

We consider five prior approaches to this problem: the unsupervised function embedding *Asm2Vec* [16], the pre-trained function embedding model *SAFE* [45], the graph-edit distance method *SMIT* [29], the matching method *CGC* of Xu et al. [62] and the N-gram method *MutantX-S* of Hu et al. [28]. More tools and techniques will be considered in the experimental evaluation, cf. Section V.

Note that in order to obtain a similarity index between programs from function embedding methods (here, *Asm2Vec* and *SAFE*), we define as the similarity metric $F(P, P') := -\sum_{x \in embeds(P)} \min_{y \in embeds(P')} \|x - y\|_2$.

TABLE I: Clone search results.

| Framework | Success | Rank of the first clone | Similarity checks runtime | Query preprocessing runtime |
|---|---|---|---|---|
| *Asm2Vec* [16] | ✗ | 11 | 1h 13m | 0s |
| *SAFE* [45] | ✗ | 9 | 10h 27m | 0s |
| *SMIT* [29] | ✗ | 52 | 59h | 0s |
| *CGC* [62] | ✗ | 11 | 2h 18m | 0s |
| *MutantX-S* [28] | ✗ | 15 | **0s** | 0s |
| *PSS* | ✓ | **1** | **0s** | **35s** |

The *VLC* core library is in the repository in three different versions built with GCC but at different optimization levels. Table I reports the results, that we compared with our own approach named *PSS*. Query prepossessing runtime does not consider the disassembly step, which is fast and common to all techniques.

As we can see, no approach but *PSS* is successful here. *SAFE* is the second best one here, ranking a clone of the target among the nine most similar programs. That is clearly not satisfactory, as it means that a human expert would have to look at nine programs before finding a real clone. Also, as expected, function-oriented similarity techniques suffer from a clear scalability issue here since they all take more than one

hour to complete the search. Moreover, the GED method and the matching method take also far too long.

Our novel *PSS* method indeed finds the clone as the most similar program. Moreover, Table I shows that while the technique has a preprocessing runtime of 35 seconds, after that the similarity checks runtime is negligible. As a result, *PSS* can scale up to large repositories with good precision.

## III. 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 [25]. 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. And, indeed, in our setting clones do have small graph-edit distances (GED). For example, this is visible in Figure 4, where the approach *GED-0* is quite accurate compared to others.

Yet, the main drawback of GED is that it is NP-hard. Worst, usual approximations have a complexity of $O(n^3)$ [59] 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* [29] is the slower of all the tested methods in the results shown in Table I, with 59 hours of computation (our method takes 35 seconds).

The spectral distance between graphs provides an interesting trade-off, as it gives a rough approximation of the graph-edit distance between graphs [61] for an affordable linear cost once eigenvalues are computed.

**Spectral Graph Analysis**. 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 [12] of G, a positive semi-definite matrix defined as follows:

$$L_{i,j} := \begin{cases} d_i & \text{if } i = j \text{ and } d_i \neq 0 \\ -1 & \text{if } i \neq j \text{ and } A_{i,j} \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

An eigenvalue $\lambda$ and its corresponding eigenvector $\vec{u}$ is a solution to the equation: $(L - \lambda I)\,\vec{u} = \vec{0}$.

The spectrum is the set $\{\lambda_1(G), \lambda_2(G), \ldots, \lambda_{|G|}(G)\}$ where $\lambda_1(G) \geq \lambda_2(G) \geq \ldots \geq \lambda_{|G|}(G)$ and where $|G|$ is the number of vertices in $G$. The enhanced Lanczos algorithm [51] computes the spectrum in time $O(dn^2)$, where $d$ is the average degree of $G$.

The starting motivation 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 to compare graph similarities. For this, we define the spectral distance between $G_1$ and $G_2$ (analogous to [32]):

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

## IV. Program Spectral Similarity (PSS)

Spectral analysis is appealing for program similarity. Yet, a program has an enormous number of instructions, or even basic blocs, that are interconnected. Applying spectral analysis to the whole control flow would be too costly. Moreover, applying spectral analysis to the call graph only is unlikely to capture critical aspects of function behaviours.

To remedy these issues, 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 propose to take advantage of this hierarchical structure to inspire a quick and stable similarity metric called *Program Spectral Similarity* (*PSS*). The *PSS* method is based on the combination of two measures.

- The first measure is the spectral distance between call graphs, including both internal and external calls. Intuitively, when a program evolves, its call graph remains relatively stable. Moreover, compiler optimizations have a negligible effect on the call graph compared to the instruction level, except for some internal functions that may be inlined;
- The second measure is a coarse spectral analysis of function control flow graphs, simply considering their number of edges.

The PSS method is then decomposed into two parts: preprocessing and similarity checks.

### A. Preprocessing

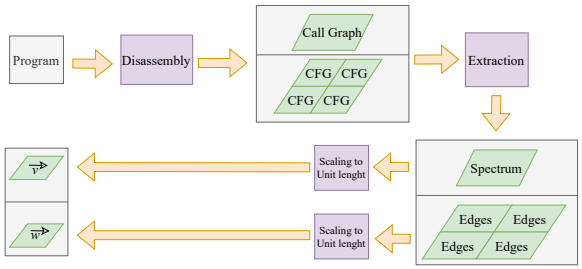



Fig. 2: *PSS* preprocessing.

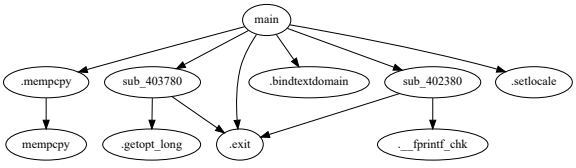



Fig. 3: A call graph.

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 3. It contains external calls such as a call to `mempcpy` as well as local functions such as `sub_403780`. Then for each local function in $P$, the preprocessing constructs its control flow graph (CFG). The set of all local function CFG is noted as $FG = \{F_1, \ldots, F_k\}$. Disassembly is implemented by running the IDA Pro disassembler v7.5 along with a script from the Kam1n0 assembly analysis platform [2]. See the recent survey of Pang et al. [53] on disassembling.

From this, we extract two key features of $P$.

- From the symmetric adjacency matrix of $CG$, 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 $F_i$ in $FG$ in descending order, that is $e_1 \geq e_2 \geq \ldots \geq e_k$, and we normalize $E$ as previously $\vec{w} := \frac{E}{\|E\|_2}$. Note that the number of edges is kind of a 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.

### B. 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 between $P_0$ and $P_1$ by averaging two measures.

The first measure is related to call graphs:

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

The second one is related to function control flow graphs:

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

where $|\vec{u}|$ is the dimension of the vector $\vec{u}$.

Then, the similarity metric $PSS$ is defined as:

$$PSS(P_0, P_1) := \frac{sCG(P_0, P_1) + sFG(P_0, P_1)}{2\sqrt{2}}$$

### C. Complexity

Recall that a repository is a database of preprocessed programs. A given unknown target program is first preprocessed, then its features are used to check its similarity index with all the preprocessed programs in the repository.

**Complexity of our method**. Graphs and Laplacian matrices are sparse in our application domain, offering quick eigenvalues computation. Nevertheless, the complexity of the query prepossessing, described in Section IV-A, 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 complexity of a similarity check, described in Section IV-B, is $O(n)$.

[2]https://github.com/McGill-DMaS/Kam1n0-Community

TABLE II: Complexity of program clone search procedures

| Framework | Class | Similarity † check | Query ‡ preprocessing |
|---|---|---|---|
| *SMIT* [29] | GED | $O(n^4)$ | $O(dn)$ |
| *CGC* [62] | Matching | $O(n^4)$ | $O(dn)$ |
| ***MutantX-S*** [28] | **N-gram** | $O(1)$ | $O(i)$ |
| *Asm2Vec* [16] | Functions ML | $O(n^2)$ | $O(n)$ |
| *Gemini* [63] | Functions ML | $O(n^2)$ | $O(n)$ |
| *SAFE* [45] | Functions ML | $O(n^2)$ | $O(n)$ |
| *$\alpha$Diff* [42] | Functions ML | $O(n^2)$ | $O(n)$ |
| *DeepBinDiff* [17] | ML | $O(n^3m^3)$ | no preproc. |
| **PSS** | **Spectral** | $O(n)$ | $O(dn^2)$ |

$n$: number of functions in the program $d$: average number of calls per function
$m$: number of basic blocks in a function $i$: number of instructions in the program
† 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 complexity of $O(n^2)$ on this problem using a direct adaptation (see Section II-B for further details). Moreover, *DeepBinDiff* [17] contains a step with a linear assignment between basic blocs with a complexity of $O(n^3m^3)$. Worse, both the graph-edit distance approximation *SMIT* [29] and the matching method of Xu et al. [62] have a complexity of $O(n^4)$. However, the complexity of *MutantX-S* [28], designed to scale up to large repositories, is only $O(1)$ – yet experiments show its precision is not fully satisfactory.

On the other hand, prior work preprocessing usually consists either in loading a call graph or in running a neural network on each function. In any case, methods whose preprocessing performs no spectral analysis have negligible preprocessing runtime. Still, as a similarity check is performed on every preprocessed program in the repository, the cost of similarity checks quickly dominates the cost of preprocessing, as highlighted in our experiments.

## V. Systematic evaluation

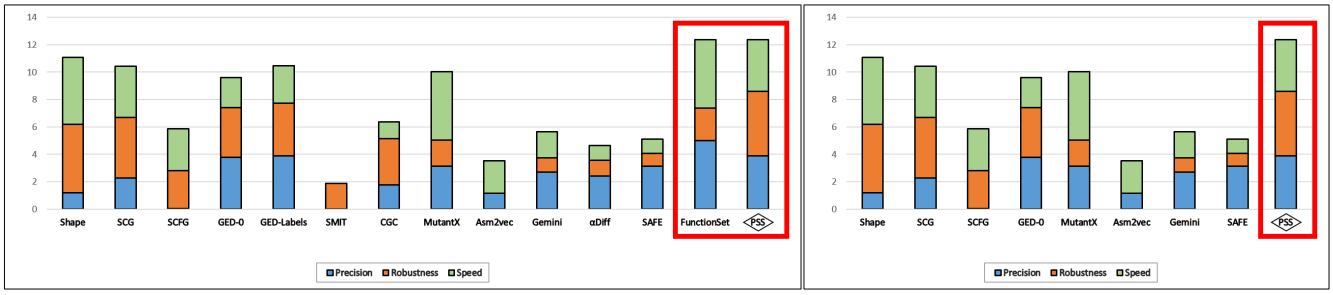
We now evaluate the potential of *PSS* in terms of precision, speed and robustness – the ability to overcome slight changes in the source code or in the compilation chain. These experiments are complemented by two other studies on IoT malware (Section VI) and large Windows repositories (Section VII).

**We consider here the following Research Questions:**

| | |
|---|---|
| RQ1 | What are the most precise methods for clone search? |
| RQ2 | What are the most robust methods for clone search? |
| RQ3 | What are the fastest methods for clone search? |
| RQ4 | What is the impact of each *PSS* component? |
| RQ5 | Does external function names enhance *PSS* ? |

### A. Summary of our main results

Before presenting the experimentation details, we highlight the key results from this study. In Figure 4, we present a plot of different frameworks over precision, robustness and speed.




(a) with external function names

(b) without external function names

Fig. 4: Precision (blue), robustness (orange), and speed (green) of all frameworks. Note that what is displayed is the inverse of the runtime. The larger the green bar, the shorter the calculation time is. (see Appendix A for methodology)

- When external function names are not available, our novel spectral method *PSS* reaches a sweet spot regarding the trade-off between precision, robustness and speed. It does not need any training phase, scales very well to large repositories and is among the most robust techniques. Therefore, it is the best candidate for intensive program clone search;
- Interestingly, when external function names are available, the function set method – which simply compares sets of external function names, is by far the fastest and most precise technique on medium-sized repositories. This is the first time such a point is highlighted in the literature, and it sheds a new light on prior work in the field. Note that in this context, PSS still ranks second and beats all other approaches relying on external function names but *FunctionSet*.

Moreover, this large study allows us to highlight that prior work in the field, mostly focused on *function-level* similarity, is not adapted to *program* clone search. Function embedding methods and graph-edit distance have relatively good precision, yet these two classes of methods are far too slow in this context. Furthermore, function embedding methods are not robust to slight variations induced by optimization levels or code versions. Matching methods are relatively slow, and while robust, they lack precision. Direct adaptations of graph-based spectral methods are fast and relatively robust but lack precision. Lastly, N-gram methods are fast and relatively precise but lack robustness.

### B. Datasets

We collect six datasets, allowing us to compare multiple methods along different optimization levels and code versions.

- **Coreutils Versions (CV):** We extract four different versions of 87 unique source codes from the popular Coreutils package for a total of 348 programs. The elapsed time between the oldest and newest versions is 13 years. Each program is compiled with optimization level $O_3$;
- **Coreutils Options (CO):** We extract from the Coreutils package a total of 104 unique source codes. Each source code is compiled with four different optimization levels ($O_0$, $O_1$, $O_2$, and $O_3$);
- **Utils Versions (UV):** On the one hand, we extract four versions of 15 unique source codes from the Binutils package. Each program is compiled with optimization level $O_2$. On the other hand, we extract four versions of 4 source codes from the Diffutils package and four versions of 3 source codes from the Findutils package. Each program is compiled with optimization level $O_3$;
- **Utils Options (UO):** We obtain 88 programs by compiling the latest versions of the precedent 22 unique source codes with 4 optimization levels;
- **Big Versions (BV):** From 16 open-source projects [3], we obtain four different versions of 21 unique source codes for a total of 84 programs. The elapsed time between the oldest and newest versions of a source code is usually four years. Programs are compiled with each project default optimization level;
- **Big Options (BO):** With the latest versions of the precedent 21 unique source codes, we obtain 84 programs by compiling them with 4 optimization levels.

Each program is given a *version level* (i.e., $V_0$, $V_1$, $V_2$ or $V_3$) depending on its relative code version.

**Compilation**. Packages Coreutils, Diffutils, and Findutils have been taken from the DeepBinDiff [17] dataset. Thus, they were compiled with GCC v5.4 on the x86 architecture. Other packages and projects were compiled using GCC v9.4 on the x86 architecture. More compilers and architectures will be considered in Sections VI and VII.

Table III presents the characteristics of each dataset (lines Repository S-* are for scenarios, cf. Section V-C). The CO and CV datasets contain hundreds of small programs, which are hard to separate as they share functionalities and origins. On the other hand, the BO and BV datasets contain less than a hundred programs from different projects that are larger. For instance, the average program in the CV dataset contains 164 functions, while the average program in the BO dataset contains 2,930 functions. As a whole, the complete dataset contains 1,108 binary programs (obtained from 277 original source codes) with a total of 811,275 functions. Note that we

[3]Open-source projects includes: *Bash*, *Coreutils*, *Dia*, *Diffutils*, *Graphviz*, *Geany*, *Git*, *Lua*, *Make*, *OpenSSH*, *OpenSSL*, *Perl*, *Ruby*, *SDL*, *SVN*, and *VLC*.

TABLE III: Datasets characteristics.

| Dataset | CV | CO | UV | UO | BV | BO |
|---|---|---|---|---|---|---|
| Size (Mo) | 18 | 22 | 82 | 91 | 114 | 97 |
| #Projets | 1 | 1 | 3 | 3 | 16 | 16 |
| # Unique source codes | 87 | 104 | 22 | 22 | 21 | 21 |
| # Options per source code | 1 | 4 | 1 | 4 | 1 | 4 |
| # Versions per source code | 4 | 1 | 4 | 1 | 4 | 1 |
| #Programs | 348 | 416 | 88 | 88 | 84 | 84 |
| Repository S-I | 87 | 104 | 22 | 22 | 21 | 21 |
| Repository S-II | 173 | 207 | 43 | 43 | 41 | 41 |
| Repository S-III | 347 | 415 | 87 | 87 | 83 | 83 |
| Average $n$ | 164 | 219 | 1222 | 1456 | 2160 | 2930 |
| Average $m$ | 9 | 7 | 19 | 18 | 12 | 11 |

$n$: Number of functions, $m$: Number of basic blocs in a function CFG
Repository S-X: Number of programs inside a repository in scenario X

will also consider much larger datasets in Sections VI and VII (up to 80k programs).

### C. Methodology

Instead of putting every dataset in a repository, we break them into **test fields** for more fine-grained results – test fields being themselves grouped into **scenarios**.

A *test field* $(T, R)$ is composed of targets $T$ and repository $R$. We note $O_i$ as (i) the optimization level $O_i$ as well as (ii) a set of programs compiled with $O_i$. For instance, the test field ($O_0$, $O_3$) from the CO dataset contains first the set of every Coreutils binary compiled with $O_0$ as targets and secondly a repository of every Coreutils binary compiled with $O_3$.

By measuring success at the level of test fields, we can study multiple *scenarios*. For instance, in an easy scenario, the repository only contains clones compiled with another optimization level. On the other hand, the repository could contain programs compiled with the same optimization level. As a result, it is harder to find a clone because sharing an optimization level increases similarities. Therefore, clone search procedures should deal with such scenarios, which is why *robustness* is a key property.

**Measures of success**. Program clone search is an *information retrieval* task. The standard evaluation metrics of information retrieval are precision and recall. Because very few programs are similar, we focus on top-1 recall.

We define the function $Recall@1(M, a, R)$, equal to one if and only if a clone of $a$ is the most similar program in the repository $R$, as ranked by the similarity metric $M$.

Let $Score(M, F)$ be the *score* of a similarity metric $M$ on test field set $F$. We take the average $Recall@1$ for every target in each test field. We define:

$$Score(M, F) := \sum_{(T,R)\in F} \sum_{a\in T} \frac{Recall@1(M, a, R \setminus \{a\})}{|T| \times |F|}$$

**Scenario I**. In the first scenario, a repository contains programs sharing the same optimization (or version) level. The corresponding target set contains programs sharing another optimization (or version) level.

We break each dataset CO, UO, and BO into 12 test fields: $(O_i, O_j), \forall\ i, j$ such that $0 \leq i, j \leq 3$ and $i \neq j$. Likewise, we break each dataset CV, UV, and BV into 12 test fields: $(V_i, V_j), \forall\ i, j$ such that $0 \leq i, j \leq 3$ and $i \neq j$. In Table III, we add the number of programs in a repository for each dataset in the first scenario (S-I).

*The first scenario is the simplest, as similarities between optimization or versions cannot trick a similarity metric.*

**Scenario II**. In the second scenario, a repository contains programs with two different optimization (or version) levels. The corresponding target set is equal to the repository.

We break each dataset CO, UO, and BO into 6 test fields: $(O_i \cup O_j, O_i \cup O_j), \forall\ i, j$ such that $0 \leq i, j \leq 3$ and $i < j$. Likewise, we break each dataset CV, UV, and BV into 6 test fields: $(V_i \cup V_j, V_i \cup V_j), \forall\ i, j$ such that $0 \leq i, j \leq 3$ and $i < j$. In Table III, we add the number of programs in a repository for each dataset in the second scenario (S-II).

*The second scenario is more challenging, as more programs are in the repository. Besides, similarities between optimization or version could trick a similarity metric.*

**Scenario III**. In the third scenario, a repository contains every program in the dataset. The corresponding target set is equal to the repository.

We transform each dataset into a unique test field. For instance, the CO dataset is transformed into the test field $(O, O)$ where $O := O_0 \cup O_1 \cup O_2 \cup O_3$. In Table III, we add the number of programs in a repository for each dataset in the third scenario (S-III).

*The third scenario is intermediate in terms of challenge. On the one hand, similarities between optimizations or version levels could trick a similarity metric. But on the other hand, there are three clones in a repository.*

**Scenario IV**. In the fourth scenario, we combine datasets CV, UV, and BV into a unique test field $(V, V)$ where $V := V_0 \cup V_1 \cup V_2 \cup V_3$. This test field has a repository of 520 programs with 130 unique source codes. Likewise, we combine datasets CO, UO, and BO into a unique test field $(O, O)$ where $O := O_0 \cup O_1 \cup O_2 \cup O_3$ This test field has a repository of 588 programs with 147 unique source codes.

*By combining our datasets, this scenario represent more computationally-demanding clone searchs (larger datasets are considered in Sections VI and VII).*

### D. Competitors

We evaluate 13 competitors, 3 baselines and a new heuristic (cf. Table IV). 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, 9 had to be reimplemented **(R)** because the original implementation was unavailable. As highlighted by Marceli et al. [44], 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. [23], we derive two methods. The first, *SCG* **(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: $SCG(X', Y') := -\sum_{i=0}^{min(|X'|,|Y'|)} |X'_i - Y'_i|$.

Likewise, we derive a method based on the control flow graph, *SCFG* **(A) (R)**. Instead of 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 [57].

Second, we implement *GED-Labels* **(A) (R)**, a computation of the GED between call graphs with labels. The algorithm is presented by Fyrbiak et al. [23]. They apply it to labeled hardware circuits. In our application, labels are sets of external function names.

Third, we implement the specific GED computation of Hu et al. [29] called *SMIT* **(R)**. It combines mnemonics similarities, matching based on external function names, and a particular Hungarian algorithm. However, we do not integrate the indexing tree of SMIT as it was designed for scenarios where many clones were present in the repository. Moreover, while the usage of an indexing tree improves efficiency, it is less precise than computing all distances.

**Matchings**. We reproduce the isomorphic test *ISO* **(R)** from Bai et al. [5]. In the algorithm, when it cannot find equivalent functions between the two programs, it either stops there or continues. We have chosen to continue. As we will see later, this choice leads the algorithm to always answer that two graphs are similar. However, stopping would have made the algorithm always answer that two graphs are different.

We also compare with the matching algorithm *CGC* **(R)** from Xu et al. [62]. This algorithm needs three parameters along with a complete classification of mnemonics. The first parameter is a threshold for mnemonics similarities. The second is a threshold for length similarity. The last is a threshold for degree similarities. After preliminary work, we set them to respectively $0.45$, $0.18$ and $0.3$.

**N-gram**. We reproduce *MutantX-S* **(R)** from the work of Hu et al. [28]. Each program is represented by the frequencies of 4-grams obtained from the opcode sequence. These frequencies are embedded into a 4096 dimensions vector by hashing.

**Function embeddings** . We are interested in evaluating binary function embeddings. Function embeddings methods allow transforming a program into a set of embedding vectors, each one representing a binary function. We define a simple similarity metric to compare programs using such sets. Let $a$ and $b$ be the two sets of embedding vectors and $F$ the similarity metric. We define: $F(a, b) := -\sum_{x \in a} \min_{y \in b} \|x - y\|_2$.

TABLE IV: Methods included in the comparison.

| Framework | Class | A | R | Similarity check | EF |
|---|---|---|---|---|---|
| $B_{size}$ | ***Baseline*** | | | $O(1)$ | |
| $D_{size}$ | ***Baseline*** | | | $O(1)$ | |
| ***Shape*** | ***Baseline*** | | | $O(1)$ | |
| ***SCG*** [23] | ***Spectral*** | × | × | $O(n)$ | |
| ***SCFG*** [23] | ***Spectral*** | × | × | $O(n)$ | |
| *GED-0* [57] | GED | × | × | $O(n^3)$ | |
| *GED-Labels* [23] | GED | × | × | $O(n^3)$ | × |
| *SMIT* [29] | GED | | × | $O(n^4)$ | × |
| *ISO* [5] | Matching | | × | $O(n^2)$ | × |
| *CGC* [62] | Matching | | × | $O(n^4)$ | × |
| ***MutantX-S*** [28] | **N-gram** | | × | $O(1)$ | |
| *Asm2Vec* [16] | Function ML | × | | $O(n^2)$ | |
| *Gemini* [63] | Function ML | × | | $O(n^2)$ | |
| *αDiff* [42] | Function ML | × | × | $O(n^2)$ | × |
| *SAFE* [45] | Function ML | × | | $O(n^2)$ | |
| *DeepBinDiff* [17] | ML | | | $O(n^3 m^3)$ | |
| ***FunctionSet*** | ***Heuristic*** | | | $O(n)$ | × |
| ***PSS*** | ***Spectral*** | | | $O(n)$ | |

A: Adapted for program clone search
R: reimplemented
EF: External function names are required

We consider *Asm2Vec* **(A)** [16]. We employ a training strategy inspired by the original paper. In fact, we compute function embeddings in the context of the second scenario. For each test field, we take programs with the highest optimization (or version) level as material for words and function embeddings for training. Other programs are only trained to produce function embeddings. Training lasts 50 epochs with an initial learning rate of $0.05$, a window size of 2, 25 negative samples for each positive sample, and the number of random walks set to 10. The final embedding size is 200.

We take *Gemini* **(A)** embedding from Xu et al. [63] in an optimistic setting. We build a version of our dataset retaining function names and employ these as ground truth. We learn over this training set for 50 epochs with five as the iterative level, a learning rate of $0.0001$, a network depth of two, and a batch size of one. Then we extract embedding vectors on usual datasets. The embedding dimension is $64$.

Moreover, we adapt the embedding of Massarelli et al. [45] with *SAFE* **(A)**. We use a pre-trained model made available by one of the authors [4]. The embedding dimension is $100$.

Lastly, we introduce *αDiff* **(A) (R)** from the framework of Liu et al. [42]. It is tailored at binary function similarity between versions. We sample 25% of the αDiff dataset [5] as our training set. We learn over this training set for 20 epochs, with a learning rate of $0.001$, and set the forgetting factor to $0.9$. The batch size of positive pairs is 100, and 200 semi-hard negative pairs are generated at each epoch. The framework αDiff proposes a distance between functions that incorporates external function names and in-out degrees in the call graphs.

**DeepBinDiff**. The framework *DeepBinDiff* from Duan et al. [17] attempts to match basic blocs between two binaries. The

[4] https://github.com/facebookresearch/SAFEtorch
[5] https://twelveand0.github.io/AlphaDiff-ASE2018-Appendix

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 research question about speed.

**Heuristic: Function set method**. Xu et al. [62] 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[6] between external function names. Let $F_a$ be the external function names set of a program $a$. The similarity metric is: $FunctionSet(a,b) := \frac{|F_a \cap F_b|}{|F_a \cup F_b|}$.

We present in Table IV the characteristics of the different methods considered here. We register the time complexity of a similarity check between two programs. We note as $n$ the number of functions and $m$ the number of basic blocs in a function CFG. We indicate whether the method requires external function names. Note that machine learning approaches require a learning phase. Moreover, *Gemini* and *GCG* require manual classification of mnemonics.

### E. RQ1: Evaluation of Precision

TABLE V: (RQ1) Scores of methods across scenarios.

| Framework | Scenario I | Scenario II | Scenario III | Average |
|---|---|---|---|---|
| $B_{size}$ | 0.28 | 0.19 | 0.26 | 0.24 |
| $D_{size}$ | 0.28 | 0.20 | 0.29 | 0.26 |
| *Shape* | 0.36 | 0.26 | 0.35 | 0.32 |
| *SCG* | 0.39 | 0.32 | 0.46 | 0.39 |
| *SCFG* | 0.10 | 0.07 | 0.12 | 0.09 |
| *GED-0* | 0.47 | 0.36 | 0.48 | 0.44 |
| *GED-Labels* | 0.49 | 0.37 | 0.54 | 0.47 |
| *SMIT* | 0.08 | 0.07 | 0.11 | 0.08 |
| *ISO* | 0.03 | 0.02 | 0.02 | 0.02 |
| *CGC* | 0.39 | 0.28 | 0.41 | 0.36 |
| *MutantX-S* | 0.49 | 0.30 | 0.46 | 0.42 |
| *Asm2Vec* | ***0.64*** | 0.15 | 0.15 | 0.31 |
| *Gemini* | 0.55 | 0.29 | 0.40 | 0.41 |
| *$\alpha$Diff* | 0.54 | 0.28 | 0.38 | 0.40 |
| *SAFE* | 0.60 | 0.27 | 0.38 | 0.42 |
| Random | 0.03 | 0.02 | 0.03 | 0.03 |
| *FunctionSet* | **0.90** | **0.62** | **0.76** | **0.76** |
| *PSS* | 0.51 | ***0.38*** | ***0.52*** | ***0.47*** |

In each column, there are two numbers in bold. The one in italic indicates the best method without using external function names, and the other one is the overall best method. For example, in scenario II, *FunctionSet* is the best method with external function names and *PSS* without. This notation will be used in all tables:
***best method without external function names*** - **overall best method**

**Methods**. First, we compute the scores of each method in the three scenarios for each dataset. Complete results are present in Tables XV, XVI, and XVII (appendix). We report average scores for each scenario in the summary Table V.

Secondly, we compute the scores of each method in the fourth scenario. We report the results in Table VI. Because of the size of the repositories (slightly more than 200 Mb), we have only evaluated the most interesting methods of each class.

**Results**. We notice that the function set method is the most powerful method in all scenarios, with a global average of $0.76$. Another method that uses external function names, *GED-Labels*, is the second-best in the second scenario with an average of $0.37$ and the third scenario with an average of $0.54$.

[6]https://en.wikipedia.org/wiki/Jaccard_index

TABLE VI: (RQ1) Scores of methods in scenario IV.

| Framework | Version | Option | Average |
|---|---|---|---|
| *Shape* | 0.24 | 0.16 | 0.20 |
| *SCG* | 0.42 | 0.23 | 0.32 |
| *SCFG* | 0.09 | 0.08 | 0.08 |
| *GED-0* | 0.43 | 0.23 | 0.33 |
| *GED-Labels* | 0.49 | 0.47 | 0.48 |
| *CGC* | 0.26 | 0.20 | 0.23 |
| *MutantX-S* | ***0.49*** | 0.18 | 0.33 |
| *Gemini* | 0.42 | 0.17 | 0.30 |
| *SAFE* | 0.35 | 0.16 | 0.26 |
| Random | 0.01 | 0.00 | 0.00 |
| *FunctionSet* | **0.67** | **0.72** | **0.70** |
| *PSS* | ***0.48*** | ***0.36*** | ***0.42*** |

***best method without external function names***
**overall best method**

On the other hand, without methods using knowledge about external function names, embedding methods perform well. For instance, the embedding method *Asm2Vec* is first in the first scenario with an average of $0.64$. Overall, our spectral method is best without access to external function names, with $0.47$ on average. It is as precise as the second-best method using external function names.

We note that in the fourth scenario, the N-gram method *MutantX-S* performs slightly better than *PSS* when searching for clones with different versions. However, our spectral method stands above *MutantX-S* by $0.18$ when searching for clones with different options. This suggests that N-gram methods do not handle well compiler optimization change.

In this scenario again, *PSS* is the best method without external function names with an average of $0.42$, while *FunctionSet* leads otherwise with an average of $0.70$.

We note that only two methods using external function names outperform other methods, namely *FunctionSet* and *GED-Labels*. Among methods using external function names, these are methods that rely the most heavily on it.

**Conclusion (RQ1)**

*PSS* globally outperforms all methods that do not use external function names. In case external function names are available, PSS obtains the second best result (even though it does not take advantage of the extra information). Finally, it is worth noticing that the very simple *FunctionSet* method clearly outperforms all competitors in this setting.

### F. RQ2: Evaluation of Robustness

We consider the robustness of a similarity metric as its resistance to the impact of shared properties such as optimization level or version level. Ideally, because the optimization level does not influence semantics, sharing optimization level should not contribute to a higher similarity index. Therefore, we will evaluate how similarities computed by each method are influenced by shared optimization or version level.

**Methods**. During each clone search, we rank each program in the repository from the most similar to the least similar,

according to a similarity metric. In the second scenario, we can split this rank into a list $A$ containing programs with the same optimization (or version) level and a list $B$ with programs containing a different optimization (or version) level. Our hypothesis $H$ is that programs are closer to the others that share the same optimization levels (or version levels). Statistically speaking, it means that programs in list $A$ have better ranks than programs in list $B$. We can compute a rank-biserial correlation. We perform this process for each clone search in the second scenario. We gather different rank-biserial correlation distributions for every framework.

TABLE VII: (RQ2) Average rank-biserial correlation for $H$.

| Framework | CV | CO | UV | UO | BV | BO |
|---|---|---|---|---|---|---|
| $B_{size}$ | 0.17 | **0.07** | **-0.02** | **0.03** | **-0.03** | **-0.04** |
| $D_{size}$ | **0.11** | **0.02** | **-0.02** | **0.06** | **-0.04** | **-0.04** |
| *Shape* | **0.15** | **0.10** | **-0.03** | **0.06** | **-0.04** | **-0.04** |
| *SCG* | **0.10** | 0.19 | **-0.01** | **0.08** | **-0.04** | **-0.04** |
| *SCFG* | 0.23 | 0.30 | **0.15** | 0.17 | **-0.02** | **-0.02** |
| *GED-0* | 0.22 | 0.25 | **-0.02** | **0.05** | **-0.04** | **-0.04** |
| *GED-Labels* | 0.16 | 0.21 | **-0.01** | **0.08** | **-0.04** | **-0.04** |
| *SMIT* | **-0.15** | **-0.58** | **-0.16** | **-0.46** | **0.00** | **-0.07** |
| *ISO* | **0.00** | **0.00** | **0.01** | **0.01** | **0.00** | **0.00** |
| *CGC* | 0.19 | 0.32 | **0.08** | **0.07** | **-0.04** | **-0.08** |
| *MutantX-S* | 0.39 | 0.63 | **0.05** | 0.28 | **-0.04** | **0.08** |
| *Asm2vec* | 0.99 | 1.00 | 0.49 | 0.65 | 0.32 | 0.45 |
| *Gemini* | 0.76 | 0.96 | 0.19 | 0.37 | **-0.04** | **0.06** |
| *$\alpha$Diff* | 0.60 | 0.93 | 0.19 | 0.33 | **-0.04** | **0.09** |
| *SAFE* | 0.81 | 0.98 | 0.20 | 0.38 | **-0.04** | **0.11** |
| Random | 0 | 0 | 0 | 0 | 0 | 0 |
| *FunctionSet* | 0.39 | 0.37 | **0.08** | 0.22 | **-0.04** | **0.02** |
| *PSS* | **0.06** | **0.13** | **0.02** | **0.09** | **-0.04** | **-0.02** |

**Average rank-biserial correlation less than 0.16**

**Results**. We report average correlations in Table VII.

In datasets CV and CO, the embedding methods *Asm2Vec*, *Gemini*, and *SAFE* provide a very high correlation for our hypothesis. For instance, clone searches done with the embedding method *Asm2Vec* attain an average correlation of $0.99$ on CV and $1.00$ on CO. The embedding method *$\alpha$Diff* provides a moderately high correlation of $0.60$ on CV but a very high correlation of $0.93$ on CO. We note that the function set method has a moderate correlation of $0.39$ on CV and $0.37$ on CO. Our spectral method has a small correlation of $0.06$ on CV and $0.13$ on CO. While, the N-gram method *MutantX-S* has a moderately high correlation of $0.39$ on CV, it has a correlation of $0.63$ on CO. Other methods provide small correlations inferior to $0.35$ for the hypothesis.

In datasets UV and UO, the embedding method *Asm2Vec* provides moderately high correlations for our hypothesis, with $0.49$ on UV and $0.65$ on UO. However, other embedding methods provide a moderate correlation. For example, *Gemini* provides a correlation of $0.19$ un UV and $0.37$ on UO. Our spectral method has a small correlation of $0.02$ on UV and $0.09$ on UO. Finally, we note that the function set method has small correlations of $0.08$ on UV and $0.22$ on UO. The rest of the method provides small correlations inferior to $0.30$.

In datasets BV and BO, the embedding method *Asm2Vec* provides a moderate correlation for our hypothesis with $0.32$ on BV and $0.45$ on BO. Our spectral method has correlations of $-0.04$ on BV and $0.02$ on BO. Other methods provide small correlations inferior to $0.11$.

We note that SMIT provides negative correlations on all datasets. We believe *Asm2Vec* to be less robust due to its specific training phase. This highlights the need to train function embeddings methods on diverse binaries.

**Conclusion (RQ2)**

*PSS* is robust to both optimization levels and program versions. Especially, it is robust even on the smaller binaries of the Coreutils and Utils datasets, where function embedding methods are confused by optimization levels and versions.

### G. RQ3: Evaluation of Speed

*The evaluation here is systematic, but on limited size benchmarks. We conduct an additional scalability analysis on much larger benchmarks (up to 80k) in Section VII.*

Because of running times, we could not experiment on DeepBinDiff [17] with our datasets. Indeed even on a dataset containing small programs, such as Coreutils Versions (CV), DeepBinDiff has an average running time of a bit more than 10 minutes per similarity check. Therefore, it would take approximately $348 \times 347 \times 630s \approx 21,130h$ to apply DeepBinDiff to the whole CV dataset. Moreover, running DeepBinDiff with *Perl* and *Code::Blocks* as inputs crashes because it would require 245 GB of memory.

TABLE VIII: (RQ3) Learning time.

| Framework | Total learning | Time per epoch |
|---|---|---|
| *Asm2Vec* | 443.8h | 2.96h |
| *Gemini* | 17.15h | 1235s |
| *$\alpha$Diff* | 58.69h | 2.93h |

We report learning time of ML-based approaches in Table VIII. Because *Asm2Vec* was trained for every test field, it has a much longer learning phase. However, the learning time per epoch to learn one element is similar to $\alpha$Diff.

TABLE IX: (RQ3) Preprocessing time.

| Framework | Total | Maximum | Average |
|---|---|---|---|
| *SCG* | 18m13s | 4m59s | 2s |
| *SCFG* | 42h37m | 36m31s | 4m42s |
| *PSS* | 18m | 4m44s | 2s |

We report the total preprocessing runtime of spectral methods in Table IX. The total preprocessing runtime is equal in every scenario because in each scenario we preprocess once every program. Other techniques preprocessing are negligible and thus not recorded. Since IDA disassembling is common to all frameworks, we do not incorporate its runtime.

While the complexity of eigenvalue decomposition is high, it takes less than twenty minutes to extract all features from every dataset with our method. On the other hand, performing eigenvalue decomposition on the complete control flow graph (*SCFG*) takes almost two days of computation, even when extracting only the top 1000 eigenvalues.

We report in Table X the average time per clone search in Scenario III, including preprossessing time. We also report the time to compute a similarity index between two programs, called a similarity check. The third scenario allows capturing

TABLE X: (RQ3) Runtimes in scenario III.

| Framework | Total | Time per Clone Search Avg (Min,Max) | Time per Similarity Check Avg (Max) |
|---|---|---|---|
| $B_{size}$ | **7s** | **0s (0s,0s)** | **0s (0s)** |
| $D_{size}$ | **6s** | **0s (0s,0s)** | **0s (0s)** |
| ***Shape*** | **1m47s** | **0s (0s,1s)** | **0s (0s)** |
| ***SCG*** | **18m14s** | **2s (0s,4m59s)** | **0s (0s)** |
| *SCFG* | 42h37m | 4m42s (0s,36m31) | 0s (0s) |
| *GED-0* | 107h20m | 12m19s (7s,5h35m) | 8s (27m44s) |
| *GED-Labels* | 61h51m | 6m57s (6s,3h55m) | 4s (5m24s) |
| *SMIT* | 4513h58m | 8h53m (19s,1100h52m) | 6m23s (43h32m) |
| ***ISO*** | **0s** | **0s (0s,0s)** | **0s (0s)** |
| *CGC* | 224h19m | 25m44s (12s,4h2m) | 18s (27m1s) |
| ***MutantX-S*** | **4s** | **0s (0s,0s)** | **0s (0s)** |
| *Asm2vec* | 87h39m | 9m50s (7s,4h10m) | 14s (22m0s) |
| *Gemini* | 134h56m | 15m30s (3s,5h57m) | 11s (17m37s) |
| *$\alpha$Diff* | 840h8m | 1h31m (2m3s,34h21m) | 1m4s (2h10m) |
| *SAFE* | 859h28m | 1h36m (1m6s,33h10m) | 1m8s (1h36m) |
| ***FunctionSet*** | **3s** | **0s (0s,0s)** | **0s (0s)** |
| ***PSS*** | **18m3s** | **2s (0s,4m44s)** | **0s (0s)** |

TABLE XI: (RQ3) Runtimes in scenario IV.

| Framework | Total | Time per Clone Search (Min,Max) | Time per Similarity Check (Min,Max) |
|---|---|---|---|
| ***Shape*** | **5m** | **(0s,2s)** | **(0s,0s)** |
| ***SCG*** | **18m16s** | **(0s,4m59s)** | **(0s,0s)** |
| *GED-0* | 487h41m | (5m52s,40h1m) | (0s,27m44s) |
| *GED-Labels* | 288h35m | (4m28s,16h45m) | (0s,5m46s) |
| *CGC* | 516h33m | (36s,14h5m) | (0s,27m1s) |
| ***MutantX-S*** | **8s** | **(0s,0s)** | **(0s,0s)** |
| *Gemini* | 322h42m | (29s,11h31m) | (0s,17m37s) |
| *SAFE* | 2135h20m | (3m58s,49h34m) | (0s,1h36m) |
| ***FunctionSet*** | **6s** | **(0s,0s)** | **(0s,0s)** |
| ***PSS*** | **18m6s** | **(0s,4m44s)** | **(0s,0s)** |

the runtime for each dataset. Again, we give equal weight to each dataset in the average computation, meaning that despite Coreutils datasets containing more binaries, their low runtime does not have a disproportionate impact. For a clone search in the third scenario, our spectral method only takes two seconds, nearly everything in the prepossessing. That is similar to the adapted spectral method for call graph *SCG*. Embedding methods take from nine minutes with *Asm2Vec* to one hour and 36 minutes with *SAFE*. Methods based on graph-edit distance do better but still require at least six minutes per clone search. The N-gram method *MutantX-S* is very fast and takes only four seconds in total, it is second to the *FunctionSet* method in terms of speed.

We report in Table XI the average time per clone search in scenario IV (2 repositories of more than 500 programs), including prepossessing time. The total runtime for *PSS* does not increase by more than three seconds compared to the runtime in scenario III. That is in sharp contrast with *SAFE*, whose runtime increases from 859 hours to 2135 hours. Section VII reports additional evidence of the scalability of *PSS* over large repositories (up to 80k programs).

**Conclusion (RQ3)**

*PSS* provides a fast similarity metric, with no learning phase and a reasonable prepossessing time. While not the fastest, our method still scales very well with repository size. Note as well that *FunctionSet* is the second fastest among all methods.

### H. RQ4: Impact of PSS components

The *PSS* method can be thought of as a combination of two components. The first component is the comparison of eigenvalues of the function call graph. The corresponding similarity metric is *simCG*. The second component is the comparison of numbers of edges from control flow graphs. The corresponding similarity metric is *simFG*. We now seek to evaluate the impact of each of these components separately.

**Results**. Table XVIII (appendix) reports global average scores across each dataset. *simFG* performs better than *simCG* on CV by $0.03$ and on CO by $0.08$. On the other hand, *simCG* is better than *simFG* on other datasets. For instance, *simCG* has a score of $0.77$ on BO while *simFG* attains $0.60$. Table XIX (appendix) reports average scores along scenarios. Still, *PSS* performs better than each component taken in isolation in every scenario, with a global average of $0.46$, while *simCG* has an average of $0.42$ and *simFG* has an average of $0.39$.

**Conclusion (RQ4)**

*PSS* is an excellent midpoint between two extrema in each scenario. On the one hand, component *simCG* is critical on the Utils and Big datasets. On the other hand, component *simFG* is key on the Coreutils datasets. Thus, we hypothesize that the numbers of edges from control flow graphs matter more in minor programs.

### I. RQ5: Impact of external function names

We have seen that the *FunctionSet* method, which merely compares the set of external function names, is really powerful. In Appendix B, we demonstrate that combining *PSS* with *FunctionSet* does not bring any improvement on this dataset.

**Conclusion (RQ5)**

On medium-sized repositories where external function names are available, *FunctionSet* is already so precise that combining it with *PSS* brings nothing.

## VI. Case Study: IoT Malware

This case study aims to compare *PSS* and other clone search methods in the setting of IoT malware classification – thus we do not consider approaches specialized in IoT malware classification [2], [3], [26], [49], [60]. We consider those competitors achieving high scalability and at least some precision in Section V, namely $B_{size}$, $D_{size}$, *Shape*, *SCG*, *MutantX-S* and *FunctionSet*.

**Dataset**. We consider 19,959 IoT malware samples from MalwareBazaar[7], submitted between March 2020 and May

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

2022. Appendix C provides more details on this dataset, which spans 8 architectures (mostly ARM, cf. Figure 6).

**Ground Truths**. Using available meta-data from antivirus reports and YARA rules, we split the data into three families of clones: 12,357 Mirai, 5,842 Gafgyt, and 1,760 Tsunami.

TABLE XII: Results on IoT malware.

| Framework | Score | Time per clone search (s) |
|---|---|---|
| $B_{size}$ | 0.821 | 0.144 |
| $D_{size}$ | 0.795 | 0.153 |
| *Shape* | 0.682 | 0.073 |
| *SCG* | 0.813 | 0.268 |
| *MutantX-S* | 0.871 | 0.778 |
| Random | 0.478 | 0 |
| *FunctionSet* | 0.624 | 0.031 |
| *PSS* | **0.888** | **0.411** |

**best method**

**Results**. In Table XII, we report scores as well as the average time per clone search in seconds. *PSS* has a score of $0.888$. As expected, *FunctionSet* does not perform well on IoT firmware, with a score of $0.624$ (only very few external names are available). Moreover, *PSS* takes only $0.411$s per clone search on a repository of 19,959 IoT malware.

**Conclusion (IoT malware)**

On IoT malware, *PSS* is the most precise method and is faster than *MutantX-S*.

## VII. Case Study: Large Repositories

The goal of this study is to demonstrate further how well *PSS* scales up to large repositories. As in the previous case study, we compare against $B_{size}$, $D_{size}$, *Shape*, *SCG*, *MutantX-S* and *FunctionSet*.

**Dataset**. We assemble the repository **R80k** of 84,992 programs running under Windows operating systems (x86, Visual Studio). This amount to more than 50 GB of raw programs. These programs range from Windows XP libraries to Windows 10 executables. There have been 30,621 security updates over twenty years. Samples are divided by **target platform** (e.g., Windows 7). We consider 10,717 applications that do not belong to a specific target platform to be part of the *common platform application* target platform. Appendix D gives more details on this dataset.

**Ground Truths**. We consider that two programs sharing both (i) file names and (ii) target platforms are clones.

**Repositories**. To study how frameworks scale, we decompose **R80k** into two additional smaller repositories: **R40k** with 42,646 programs, and **R20k** with 21,113 programs. Our decomposition guarantees that each repository contains either all clones of a program or no clones of this program. In total, 49,443 programs have a clone (**R40k**: 24,983, **R20k**: 12,083).

**Precision**. In Table XIII, we present the scores of each framework. *PSS* has the best score on all three repositories, with $0.4736$ on **R80k**. Unexpectedly, *FunctionSet* has low scores here, e.g. only $0.4255$ on **R80k**. Actually, *FunctionSet* is second in terms of precision on **R20k**, but third on **R40k** and fifth on **R80k**. Actually, it turns out that, as the number of programs increases, combinations of API calls are more and more disconnected from similarity – intuitively, the number of combinations of API calls increases less than the number of programs at some point. On **R20k**, $50.80\%$ of queries have another program with the same API call combination in the repository (**R40k**: $55.49\%$, **R80k**: $61.77\%$). Also, while on **R20k** $18.59\%$ of the equalities between API call combinations correspond to program clones, it drops to $8.64\%$ on **R40k** and $4.67\%$ on **R80k**.

TABLE XIII: Scores on large repositories.

| Framework | Score on **R80k** | Score on **R40k** | Score on **R20k** |
|---|---|---|---|
| $B_{size}$ | 0.1955 | 0.2593 | 0.3122 |
| $D_{size}$ | 0.4447 | 0.4719 | 0.4616 |
| *Shape* | 0.3873 | 0.4507 | 0.4808 |
| *SCG* | 0.4418 | 0.4971 | 0.5092 |
| *MutantX-S* | 0.4731 | 0.5316 | 0.5521 |
| Random | 0.0001 | 0.0002 | 0.0002 |
| *FunctionSet* | 0.4255 | 0.4979 | 0.5521 |
| *PSS* | **0.4736** | **0.5406** | **0.5705** |

**best method**

TABLE XIV: Time per clone search on large repositories

| Repositories | **R80k** | | | **R40k** | | | **R20k** | | |
|---|---|---|---|---|---|---|---|---|---|
| Framework | Total | QP | SC | Total | QP | SC | Total | QP | SC |
| $B_{size}$ | 0.58 | 0.00 | 0.58 | 0.30 | 0.00 | 0.30 | 0.17 | 0.00 | 0.17 |
| $D_{size}$ | 0.58 | 0.00 | 0.58 | 0.30 | 0.00 | 0.30 | 0.15 | 0.00 | 0.15 |
| *Shape* | 0.28 | 0.00 | 0.28 | 0.14 | 0.00 | 0.14 | 0.07 | 0.00 | 0.07 |
| *SCG* | 14.05 | 13.02 | 1.03 | 13.60 | 13.08 | 0.52 | 13.31 | 13.05 | 0.26 |
| *MutantX-S* | 2.96 | 0.00 | **2.96** | 1.57 | 0.00 | **1.57** | 0.76 | 0.00 | **0.76** |
| *FunctionSet* | 2.01 | 0.00 | 2.01 | 1.00 | 0.00 | 1.00 | 0.50 | 0.00 | 0.50 |
| *PSS* | 15.51 | 13.42 | **2.09** | 14.53 | 13.49 | **1.04** | 13.97 | 13.45 | **0.52** |

Time in seconds, QP: Query Preprocessing, SC: Similarity Checks

**Scaling**. Table XIV presents the average time per clone search on each repository. *PSS* takes $13.97$s per clone search on **R20k**, while *MutantX-S* takes $0.76$s. However, *PSS* spends $13.45$s on query preprocessing and $0.52$s on performing similarity checks. On **R40k**, whose size is 2x the size of **R20k**, *PSS* takes $14.53$s per clone search with $13.49$s on query preprocessing. The time spent on query preprocessing is stable because it depends on the data distribution of targets. On the other hand, the average time spent on similarity checks for each framework is linear to the repository size, and 50% slower for *Mutant-X-S* compared to *PSS*. In Figure 13 (appendix), we perform linear regressions of average time per clone search w.r.t. repository size. We estimate here that *PSS* would be faster than *MutantX-S* for repository sizes superior to 1.5 million samples.

**Conclusion (Large Repositories)**

*PSS* is the most precise method on huge repositories of Windows programs, while *FunctionSet* has poor precision. Moreover, *PSS* shows excellent scaling abilities due to its very fast similarity checks.

## VIII. Related works

Binary code similarities are extensively studied. As a testimony, the review of Haq and Caballero [27] reports numerous input and output granularities on which to study similarities.

### A. Pioneering approaches

The pioneering work of Dullien in 2004 [18] introduced a graph-based program diffing approach that constructs a call graph isomorphism. A follow-up [19] 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. [37] 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 [24] 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. However, symbolic execution and theorem proving are expensive.

### B. Program similarity

The few recent works about program-level similarity [47], [64] have already been thoroughly discussed. Still, we can mention a few more approaches.

N-gram methods compare instruction sequences [28], [33], [48], [58]. While we could have employed more fine-grained methods than *MutantX-S* [28] – for example Exposé [48] 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 [4], [31], [41], [46]. Nevertheless, it is hard to thoroughly explore the execution space with dynamic traces – leading to poor precision, and handling large code repositories requires to automate the task of detecting the sources of input and output of all programs in the repository, which can be very complicated.

Bruschi et al. [9] 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. [43] 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 [7], [52].

We have already discussed matching approaches [5], [62]. While Xu et al. [62] studied a metric that compared function mnemonics and external function names, they found that their more complex matching was better. On the other hand, we found that simplifying it by comparing only the set of external function names is better than their matching. A few other matching approaches have been proposed [8], [39], sharing the same strengths and weaknesses.

### C. Function similarity

The last five years have seen a tremendous increase in the popularity of binary function similarity with machine learning [16], [42], [45], [63], [65]. 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 [10] analyzes various execution traces with concepts such as pruning. The work of Hu et al. [30] emulates binary functions to create semantic signatures. Pewny et al. [54] 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 [14], [38], [55] perform simplification before comparing. In FirmUp [13], 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 [34] extracts statistical features at various granularities, while discovRE [20] and Genius [21] extract features such as the number of arithmetic instructions. Note that Gemini [63] leverages static features from Genius into a machine learning framework.

### D. Graph similarity

A key question in program similarity is how to compare graphs efficiently. New suggestions for graph similarities include novel graph kernels [22], [36], [50] and the use of machine learning to approximate intractable properties such as graph-edit distance [6], [40], [56]. 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.

## IX. Conclusion

We consider the problem of searching program clones in large code repositories. While many works have been devoted to function similarity, the few existing techniques for program similarity suffer either from scalability issues over such large code bases, or low precision or low robustness to slight code variations. We propose a novel method called Program Spectral Similarity (*PSS*) that reaches a sweet spot in terms of precision, speed and robustness. Especially, PSS is shown to perform much better than prior approaches in the program clone search setting.

Interestingly, while we are primarily interested in a setup where external function names are not available (e.g., static libraries, lightweight obfuscation), we show that when they are available, a simple distance between sets of external function names beats by far all other approaches considered on medium-sized repositories of standard Linux binaries, *PSS* being still the second one in that setting.

This work opens up several immediate questions, including in particular the following two that seem promising. First,

spectral analysis provides a set of methods that may enhance clone search heuristics by adding extra features. Second, external function detection seems to have been underestimated and may be "a fast and dirty" approach for program clustering.

## REFERENCES


[1] H. A. B. Ahmed, A.-O. Boudraa, and D. Dare-Emzivat, "A joint spectral similarity measure for graphs classification," *Pattern Recognition Letters*, 2019.

[2] H. Alasmary, A. Khormali, A. Anwar, J. Park, J. Choi, A. Abusnaina, A. Awad, D. Nyang, and D. Mohaisen, "Analyzing and detecting emerging internet of things malware: A graph-based approach," *IEEE Internet of Things Journal*, 2019.

[3] M. Alhanahnah, Q. Lin, Q. Yan, N. Zhang, and Z. Chen, "Efficient signature generation for classifying cross-architecture iot malware," in *2018 IEEE Conference on Communications and Network Security (CNS)*, 2018.

[4] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis," *Journal in Computer Virology*, 2011.

[5] J. Bai, Q. Shi, and S. Mu, "A malware and variant detection method using function call graph isomorphism," *Security and Communication Networks*, 2019.

[6] Y. Bai, H. Ding, S. Bian, T. Chen, Y. Sun, and W. Wang, "Simgnn: A neural network approach to fast graph similarity computation," in *Proceedings of the 12th ACM International Conference on Web Search and Data Mining*, 2019.

[7] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016.

[8] M. Bourquin, A. King, and E. Robbins, "Binslayer: Accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013.

[9] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2006.

[10] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[11] B. Cheng, J. Ming, E. A. Leal, H. Zhang, J. Fu, G. Peng, and J.-Y. Marion, "Obfuscation-Resilient executable payload extraction from packed malware," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[12] F. Chung, *Spectral graph theory*. American Mathematical Society, 1997.

[13] Y. David, N. Partush, and E. Yahav, "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*, 2018.

[14] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[15] P. Deshpande and M. Stamp, "Metamorphic malware detection using function call graph analysis," *MIS Review*, 2016.

[16] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," *IEEE Symposium on Security and Privacy (SP)*, 2019.

[17] Y. Duan, X. Li, J. Wang, and H. Yin, "Learning program-wide code representations for binary diffing," in *27th Network and Distributed System Security Symposium*, 2020.

[18] T. Dullien, "Structural comparison of executable objects," *Workshop on Detection of Intrusions and Malware & Vulnerability Assessment*, 2004.

[19] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," *SSTIC*, 2005.

[20] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code," in *NDSS*, 2016.

[21] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[22] A. Feragen, N. Kasenburg, J. Petersen, M. de Bruijne, and K. Borgwardt, "Scalable kernels for graphs with continuous attributes," *26th International Conference on Neural Information Processing Systems*, 2013.

[23] M. Fyrbiak, S. Wallat, S. Reinhard, N. Bissantz, and C. Paar, "Graph similarity and its applications to hardware security," *IEEE Transactions on Computers*, 2020.

[24] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Proceedings on International Conference on Information and Communications Security*, 2008.

[25] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance," *Pattern Analysis and Applications*, 2010.

[26] H. HaddadPajouh, A. Dehghantanha, R. Khayami, and K.-K. R. Choo, "A deep recurrent neural network based approach for internet of things malware threat hunting," *Future Generation Computer Systems*, 2018.

[27] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Computing Surveys (CSUR)*, 2021.

[28] X. Hu, S. Bhatkar, K. Griffin, and K. G. Shin, "Mutantx-s: Scalable malware clustering based on static features," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, 2013.

[29] X. Hu, T. cker Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2009.

[30] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017.

[31] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Proceedings of the 22nd USENIX Conference on Security*, 2013.

[32] I. Jovanović and Z. Stanić, "Spectral distances of graphs," *Linear Algebra and its Applications*, 2012.

[33] B. Kang, T. Kim, H. Kwon, Y. Choi, and E. G. Im, "Malware classification method via binary content comparison," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, 2012.

[34] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[35] O. Kostakis, J. Kinable, H. Mahmoudi, and K. Mustonen, "Improved call graph comparison using simulated annealing," in *Proceedings of the ACM Symposium on Applied Computing*, 2011.

[36] N. M. Kriege, P.-L. Giscard, and R. C. Wilson, "On valid optimal assignment kernels and applications to graph classification," in *30th International Conference on Neural Information Processing Systems*, 2016.

[37] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," *International Workshop on Recent Advances in Intrusion Detection*, 2006.

[38] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice'," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013.

[39] Y. R. Lee, B. Kang, and E. G. Im, "Function matching-based binary-level software similarity calculation," in *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, 2013.

[40] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *36th International conference on machine learning*, 2019.

[41] M. Lindorfer, A. Di Federico, F. Maggi, P. M. Comparetti, and S. Zanero, "Lines of malicious code: Insights into the malicious software industry," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[42] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "$\alpha$diff: Cross-version binary code similarity detection with dnn," in *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[43] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, 2017.

[44] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, and Y. Fratantonio, "How machine learning is solving the binary function similarity problem," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[45] L. Massarelli, G. A. D. Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Function representations for binary similarity," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[46] J. Ming, D. Xu, Y. Jiang, and D. Wu, "Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *USENIX Security Symposium*, 2017.

[47] J. Ming, D. Xu, and D. Wu, "Memoized semantics-based binary diffing with application to malware lineage inference," in *IFIP Advances in Information and Communication Technology*, 2015.

[48] B. H. Ng and A. Prakash, "Expose: Discovering potential binary code re-use," in *37th IEEE Annual Computer Software and Applications Conference*, 2013.

[49] H.-T. Nguyen, Q.-D. Ngo, and V.-H. Le, "A novel graph-based approach for iot botnet detection," *International Journal of Information Security*, 2020.

[50] G. Nikolentzos, P. Meladianos, S. Limnios, and M. Vazirgiannis, "A degeneracy framework for graph similarity," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI-18*, 2018.

[51] I. Ojalvo and M. Newman, "Vibration modes of large structures by an automatic matrix-reduction method," *Aiaa Journal - AIAA J*, 1970.

[52] M. Ollivier, S. Bardin, R. Bonichon, and J.-Y. Marion, "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*, 2019.

[53] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "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)*, 2021.

[54] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *IEEE Symposium on Security and Privacy*, 2015.

[55] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[56] P. Riba, A. Fischer, J. Lladós, and A. Fornés, "Learning graph edit distance by graph neural networks," *Pattern Recognition*, 2021.

[57] A. Sanfeliu and K.-S. Fu, "A distance measure between attributed relational graphs for pattern recognition," *IEEE Transactions on Systems, Man, and Cybernetics*, 1983.

[58] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-sequence-based malware detection," in *Engineering Secure Software and Systems*, F. Massacci, D. Wallach, and N. Zannone, Eds., 2010.

[59] F. Serratosa, "Fast computation of bipartite graph matching," *Pattern Recognition Letters*, 2014.

[60] J. Su, D. V. Vasconcellos, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai, "Lightweight classification of iot malware based on image recognition," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, 2018.

[61] R. C. Wilson and P. Zhu, "A study of graph spectra for comparing graphs and trees," *Pattern Recognition*, 2008.

[62] M. Xu, L. Wu, S. Qi, J. Xu, H. Zhang, Y. Ren, and N. Zheng, "A similarity metric method of obfuscated malware using function-call graph," *Journal of Computer Virology and Hacking Techniques*, 2013.

[63] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "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*, 2017.

[64] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: Security patch analysis for binaries towards understanding the pain and pills," in *IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017.

[65] J. Yang, C. Fu, X.-Y. Liu, H. Yin, and P. Zhou, "Codee: A tensor embedding scheme for binary code search," *IEEE Transactions on Software Engineering*, 2021.

## APPENDIX

TABLE XV: Scores of methods in scenario I.

| Framework | CV | CO | UV | UO | BV | BO |
|---|---|---|---|---|---|---|
| $B_{size}$ | 0.07 | 0.06 | 0.42 | 0.36 | 0.31 | 0.48 |
| $D_{size}$ | 0.07 | 0.04 | 0.37 | 0.38 | 0.34 | 0.49 |
| *Shape* | 0.05 | 0.04 | 0.46 | 0.28 | 0.58 | 0.72 |
| *SCG* | 0.15 | 0.07 | 0.54 | 0.36 | 0.63 | 0.58 |
| *SCFG* | 0.12 | 0.08 | 0.07 | 0.07 | 0.13 | 0.10 |
| *GED-0* | 0.14 | 0.11 | 0.58 | 0.45 | 0.72 | ***0.82*** |
| *GED-Labels* | 0.28 | 0.25 | 0.55 | 0.34 | 0.71 | 0.78 |
| *SMIT* | 0.03 | 0.01 | 0.09 | 0.09 | 0.14 | 0.10 |
| *ISO* | 0.01 | 0.01 | 0.06 | 0.06 | 0.02 | 0.04 |
| *CGC* | 0.12 | 0.12 | 0.61 | 0.44 | 0.58 | 0.48 |
| *MutantX-S* | 0.43 | 0.14 | 0.63 | 0.35 | 0.83 | 0.53 |
| *Asm2vec* | 0.49 | 0.38 | ***0.75*** | ***0.57*** | 0.85 | 0.77 |
| *Gemini* | ***0.55*** | 0.35 | 0.57 | 0.38 | 0.87 | 0.59 |
| *$\alpha$Diff* | 0.24 | 0.26 | 0.74 | 0.61 | 0.85 | 0.56 |
| *SAFE* | 0.51 | ***0.41*** | 0.74 | 0.42 | ***0.91*** | 0.64 |
| Random | 0.01 | 0.01 | 0.05 | 0.05 | 0.05 | 0.05 |
| *FunctionSet* | **0.85** | **0.86** | **0.90** | **0.91** | **0.95** | **0.96** |
| *PSS* | 0.28 | 0.25 | 0.57 | 0.43 | 0.78 | 0.73 |

***best method without external function names***
**best method overall**

TABLE XVI: Scores of methods in scenario II.

| Framework | CV | CO | UV | UO | BV | BO |
|---|---|---|---|---|---|---|
| $B_{size}$ | 0.04 | 0.04 | 0.30 | 0.25 | 0.18 | 0.32 |
| $D_{size}$ | 0.04 | 0.02 | 0.27 | 0.24 | 0.27 | 0.38 |
| *Shape* | 0.03 | 0.01 | 0.27 | 0.12 | 0.51 | 0.65 |
| *SCG* | 0.10 | 0.03 | ***0.39*** | ***0.30*** | 0.58 | 0.53 |
| *SCFG* | 0.08 | 0.03 | 0.06 | 0.03 | 0.12 | 0.10 |
| *GED-0* | 0.11 | 0.02 | 0.35 | 0.28 | 0.67 | ***0.75*** |
| *GED-Labels* | 0.16 | 0.19 | 0.40 | 0.23 | 0.62 | 0.65 |
| *SMIT* | 0.02 | 0.00 | 0.08 | 0.07 | 0.14 | 0.10 |
| *ISO* | 0.01 | 0.01 | 0.03 | 0.03 | 0.02 | 0.02 |
| *CGC* | 0.05 | 0.04 | 0.39 | 0.24 | 0.51 | 0.42 |
| *MutantX-S* | 0.14 | 0.01 | 0.37 | 0.13 | 0.79 | 0.38 |
| *Asm2vec* | 0.01 | 0.00 | 0.03 | 0.02 | 0.45 | 0.37 |
| *Gemini* | 0.15 | 0.01 | 0.23 | 0.05 | ***0.83*** | 0.46 |
| *$\alpha$Diff* | 0.06 | 0.00 | 0.32 | 0.16 | 0.74 | 0.41 |
| *SAFE* | 0.08 | 0.01 | 0.25 | 0.06 | 0.81 | 0.42 |
| Random | 0.01 | 0.00 | 0.02 | 0.02 | 0.02 | 0.02 |
| *FunctionSet* | **0.39** | **0.52** | **0.55** | **0.52** | **0.87** | **0.89** |
| *PSS* | ***0.16*** | ***0.10*** | 0.36 | 0.28 | 0.75 | 0.63 |

***best method without external function names***
**best method overall**

### A. Global evaluation

We consider that a higher average score noted in Table V induces a better precision. Moreover, we assume that lower absolute correlations for hypothesis $H$, noted in Table VII, induce robustness. Finally, we consider that a low total running time in scenario III, noted in Table X, induces speed. Each dimension is then transformed using five quartiles to a uniform distribution.

TABLE XVII: Scores of methods in scenario III.

| Framework | CV | CO | UV | UO | BV | BO |
|---|---|---|---|---|---|---|
| $B_{size}$ | 0.10 | 0.10 | 0.35 | 0.36 | 0.25 | 0.39 |
| $D_{size}$ | 0.06 | 0.04 | 0.33 | 0.39 | 0.37 | 0.58 |
| *Shape* | 0.05 | 0.01 | 0.38 | 0.20 | 0.71 | 0.76 |
| *SCG* | 0.29 | 0.06 | 0.45 | ***0.42*** | 0.77 | 0.76 |
| *SCFG* | 0.22 | 0.07 | 0.08 | 0.06 | 0.18 | 0.18 |
| *GED-0* | 0.31 | 0.04 | 0.41 | 0.41 | 0.83 | ***0.87*** |
| *GED-Labels* | 0.38 | 0.43 | 0.53 | 0.32 | 0.80 | 0.80 |
| *SMIT* | 0.06 | 0.01 | 0.09 | 0.09 | 0.23 | 0.15 |
| *ISO* | 0.00 | 0.01 | 0.03 | 0.02 | 0.04 | 0.01 |
| *CGC* | 0.12 | 0.10 | 0.49 | 0.38 | 0.71 | 0.65 |
| *MutantX-S* | 0.38 | 0.02 | ***0.48*** | 0.30 | 0.92 | 0.67 |
| *Asm2vec* | 0.00 | 0.00 | 0.02 | 0.02 | 0.42 | 0.45 |
| *Gemini* | 0.31 | 0.04 | 0.30 | 0.12 | ***0.95*** | 0.68 |
| *$\alpha$Diff* | 0.19 | 0.00 | 0.38 | 0.25 | 0.87 | 0.62 |
| *SAFE* | 0.20 | 0.02 | 0.32 | 0.12 | 0.94 | 0.64 |
| Random | 0.01 | 0.01 | 0.03 | 0.03 | 0.04 | 0.04 |
| *FunctionSet* | **0.61** | **0.69** | **0.65** | **0.69** | **0.96** | **0.92** |
| *PSS* | ***0.39*** | ***0.24*** | 0.39 | ***0.42*** | 0.90 | 0.77 |

***best method without external function names***
**overall best method**

TABLE XVIII: (RQ4) Scores of *PSS* components across datasets.

| Component | CV | CO | UV | UO | BV | BO |
|---|---|---|---|---|---|---|
| *simCG* | 0.20 | 0.07 | ***0.45*** | ***0.41*** | ***0.77*** | ***0.77*** |
| *simFG* | ***0.23*** | ***0.15*** | 0.40 | 0.28 | 0.71 | 0.60 |
| *PSS* | **0.28** | **0.20** | 0.44 | 0.38 | **0.81** | 0.71 |

***best component***, **best metric**

TABLE XIX: (RQ4) Scores of *PSS* components across scenarios.

| Component | Scenario I | Scenario II | Scenario III | Scenario IV | Average |
|---|---|---|---|---|---|
| *simCG* | ***0.47*** | ***0.37*** | ***0.49*** | 0.35 | ***0.42*** |
| *simFG* | 0.40 | 0.31 | 0.47 | ***0.37*** | 0.39 |
| *PSS* | **0.51** | **0.38** | **0.52** | **0.42** | **0.46** |

***best component***, **best metric**

### B. RQ5: Impact of external function names

We have seen that the *FunctionSet* method, which merely compares the set of external function names, is really powerful. We investigate if we can combine *PSS* with *FunctionSet*.

**Methods**. We define the new similarity metric *SFS*, as:

$$SFS(\alpha, a, b) := \alpha FunctionSet(a, b) + (1 - \alpha)PSS(a, b)$$

*SFS* goes from *FunctionSet* to *SFS* when $\alpha$ increases. We compute the scores of *SFS* in relation to $\alpha$.

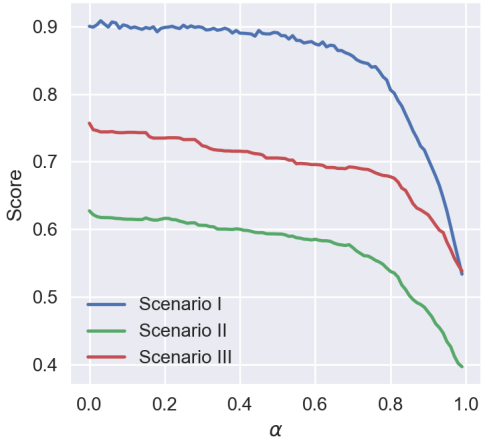



Fig. 5: Score of $SFS$ along scenarios when $\alpha$ increases.

**Results**. Figure 5 shows that, in the first three scenarios, the average score is reduced when $\alpha$ increases. The three curves are close to decreasing monotonously. There is no trade-off leading to a significant advantage over *FunctionSet*.

### C. IoT case study

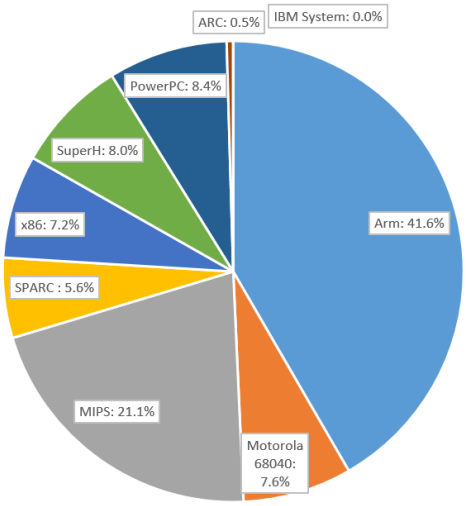



Fig. 6: Architectures of IoT malware.

The preprocessing of *SCG* took 1152s in total, while the preprocessing of *PSS* took 1002s. For each architecture, *MutantX-S* requires writing a specific algorithm that extracts OPCode bytes from an instruction.

### D. Large repositories case study

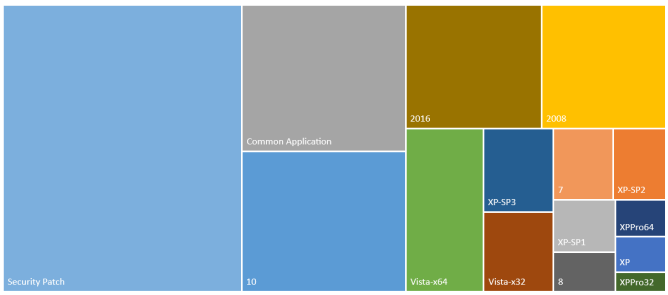



Fig. 7: Distribution of target platforms.

Figure 7 shows that most programs are security patches seconded by common platform applications. Then come platform-specific programs from Windows 10 down to Windows XP.

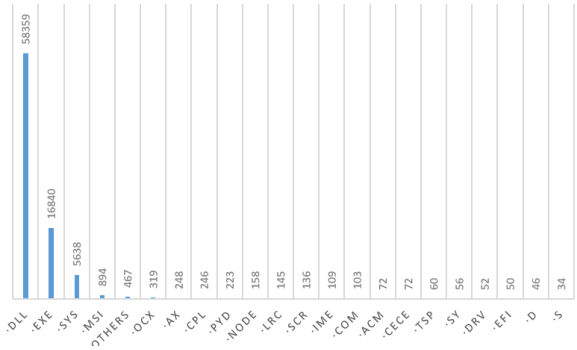



Fig. 8: Distribution of file extensions.

Figure 8 shows that most programs are dynamic libraries seconded by executables.

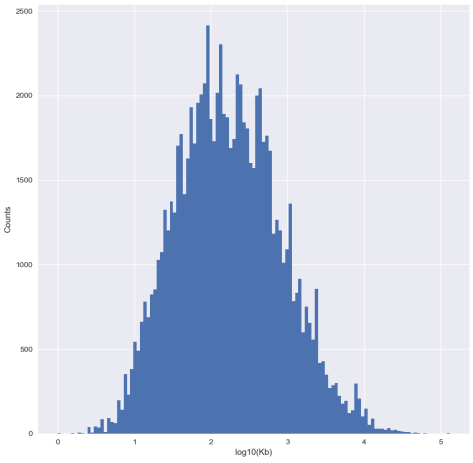



Fig. 9: Histogram of log10 of file kilobytes.

Figure 9 shows program sizes span from a few kilobytes up to a thousand megabytes. Likewise, Figure 10 illustrates that programs contain call graphs with a few dozen up to 100,000 nodes.

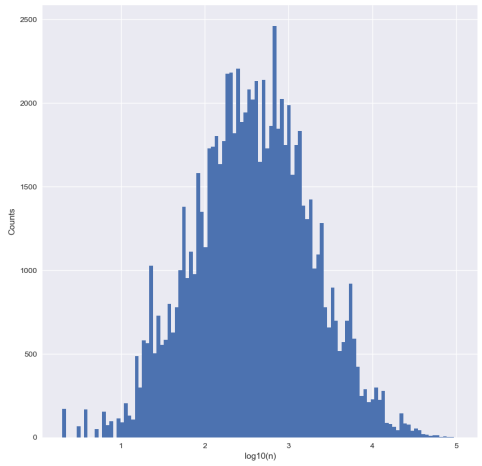



Fig. 10: Histogram of log10 of call graphs number of nodes.

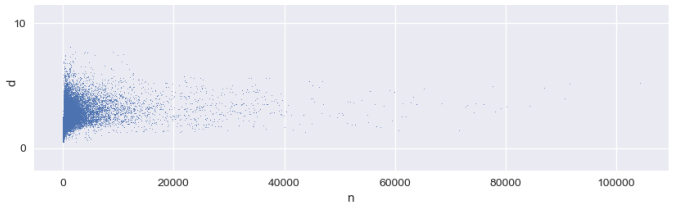



Fig. 11: A restricted plot of call graphs average degree against their number of nodes.

Figure 11 shows that call graphs are sparse by plotting the average degree against the number of nodes. The vast majority of call graphs have an average degree of less than ten. Furthermore, the average degree does not increase with the number of nodes.

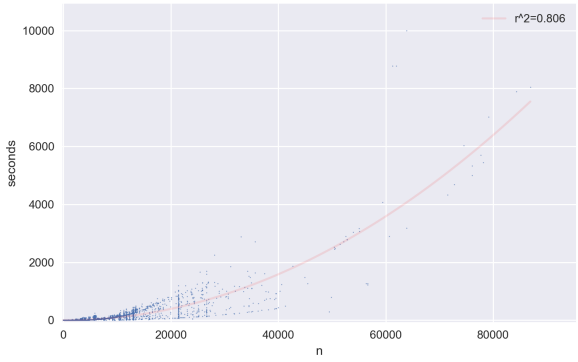



Fig. 12: Plot of query preprocessing runtimes of *PSS* against call graphs number of nodes. The red line is a second-order polynomial regression with ordinary least squares.

Figure 12 shows the link between query preprocessing runtime and call graph number of nodes. A regression with a degree two polynomial achieves a high coefficient of determination of $0.806$. Remind that, as seen in the last paragraph, the average degree $d$ is constant with $n$. Hence, Lanczos algorithm running time, which has complexity $O(dn^2)$, is well predicted by a degree two polynomial.

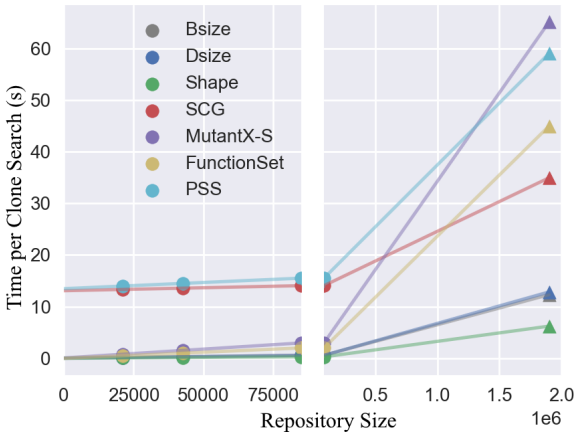



Fig. 13: Scaling capacity of frameworks.

Figure 13 shows for each framework the average time per clone search in relation to the repository size. Using linear regressions, we estimate that *PSS* would be faster than *MutantX-S* on repository sizes superior to 1.5 million samples.