Assessing the Effects of Source Language on Binary Similarity Tools

Landen Doty
landoty@ku.edu
University of Kansas
Lawrence, Kansas, USA

Dr. Prasad Kulkarni
prasadk@ku.edu
University of Kansas
Lawrence, Kansas, USA

ABSTRACT

State-of-the-art binary similarity tools have been developed to account for and are evaluated against variations in compilers, compiler flags, optimization levels, architectures, and even obfuscations. Although these tools aim to measure and detect binary code segments generated from similar or identical source code segments, they have yet to be evaluated on source languages other than C/C++.

We present a **work in progress** to assess the effects of source language on modern binary similarity tools. Specifically, we provide a comparative investigation on the efficacy of BSim, a recently released component of the Ghidra framework, when comparing binaries produced by C as well as Rust. Using a benchmark of 800 binaries and more than one million functions, we investigate the overall accuracy and differentiating ability of BSim and find that the source language introduces a significant degree of imprecision not previously documented. We also provide a technical overview of the BSim utility, which provides context for our assessment results and a clear direction for addressing the shortcomings highlighted by our findings.

CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**; • **General and reference** → *Evaluation.*

KEYWORDS

Binary Code, Diffing, Similarity, Ghidra, BSim, Rust

1 INTRODUCTION

Binary code similarity is a fundamental technique to compare a pair, set, or large corpus of binary-level code segments - instruction sequences, basic blocks, functions, etc. Binary code comparison has applications in several domains, including malware classification and detection [2, 6, 10, 16], vulnerability research [8, 17, 20], and even intellectual property protection [13]. Given its real-world applicability and the growing number of programs distributed without source, binary code similarity has seen a sustained research effort to both improve existing techniques and develop novel approaches.

Early binary similarity work, such as BinDiff, leveraged structural representations of binary code, such as control flow, data flow, and call graphs [4, 5]. However, *binary*-level comparison of software components is uniquely challenging, as even small variations in the compilation environment, including the compiler, optimization levels, and various compiler flags, can result in significant changes in the generated machine code. As such, more recent approaches attempt to capture semantic understanding of binary code segments using traditional program analysis techniques such as symbolic execution and fuzzing, and have shown improved results across variations of compiler settings [7, 15, 20]. Further, the use of machine learning and data science techniques - natural language processing (NLP), approximate nearest neighbor, and deep learning - has shown promise in improving the state of the art [3, 14, 19].

Despite the vast number of research endeavors focused on this area, to our knowledge, no current work has investigated how *source*-level constructs affect the efficacy of *binary*-level similarity techniques. With the growing adoption of non-C/C++ high-level compiled languages, practitioners will certainly see an increasing amount of binary artifacts compiled from richer high-level abstractions than those seen in C/C++. Thus, if the tools and frameworks available are overfit to patterns most commonly seen in C/C++ binaries, practitioners may be at a distinct disadvantage when comparing binaries from languages like Rust and Golang.

This paper presents a first in-depth analysis of the effects of source language on binary code similarity techniques. With a specific interest in tools readily available to binary analysis practitioners, we conduct our investigation on *BSim*, a recently open-sourced binary similarity tool in the Ghidra Software Reverse Engineering Framework developed by the National Security Agency (NSA). As a **Work In Progress** submission, this paper includes 1) a baseline analysis of BSim's performance on C/C++ binaries, 2) a comparative analysis on *Rust* binaries and 3) early results and discussion towards addressing the observed effects.

Beyond this investigation, this paper also includes a high-level technical description of BSim, its components, and configurations resulting from our analysis of its open-source content. This information is important in contextualizing the results of our assessments and provides future work with a baseline of technical understanding.

In summary, this work makes the following contributions:

- We provide a novel investigation of the effects of source language on binary similarity. Using Rust as our experimental language, we show that an existing tool, BSim, is tuned to features seen in C-based languages and its performance is degraded when comparing binaries not of C origin.
- We detail the internal components of BSim, its feature representation, and available configuration and tuning capabilities. This contribution results from our manual analysis of BSim's released source code as well as extensive work

HotSoS '25, April 1–3, 2025, Nashville, TN

```
bl0:
 mov eax, [0x14813c9c]
 test eax, eax
 jnz bl2;
bl1:
 call func1
 mov [0x14813c9c], eax
bl2:
 ret
```

Figure 1: Feature Representation Code Snippet

using BSim to facilitate our assessment of source language effects.

2 BACKGROUND: BSIM OVERVIEW

Built atop the Ghidra Software Reverse Engineering framework, BSim - short for *Behavioral Similarity* - is a binary similarity database technology inspired by current text retrieval techniques. It aims to be "tolerant of variation" in compiler configurations by normalizing certain components in binary code, such as storage location and instruction ordering. These components, and others, are sources of common inconsistencies in prior binary code similarity tools due to variation in compiler versions, optimization passes, etc. To further account for variation in a function's binary representation, BSim performs *nearest neighbor* queries via **Locality Sensitive Hashing (LSH)**, providing users a configurable set of probable matches. We refer the reader to the official Ghidra documentation for a step-by-step procedure for creating, populating, and querying BSim databases [1]. Thus, the remainder of this section details components of BSim *not* explicitly included in the official documentation. From our analysis of BSim's open source content, we have documented high-level functionality in its feature representation, comparison metrics, and database configuration.

2.1 Feature Representation

To succinctly capture the behavioral features of a function from its binary-level representation, BSim utilizes components of the *control flow graph (CFG)* and *abstract syntax tree (AST)* generated from Ghidra's intermediate representation (IR), *p-code*. Using an IR such as *p-code* allows BSim to normalize variation in instructions, register allocations, etc. which are often affected when toggling compiler options and versions. Using this scheme, BSim characterizes binary code features as either `ControlFlow` or `DataFlow`, as well as specialized `Combined` and `DualFlow` features which combine both control flow and data flow information.

Clearly, it is not sufficient to store only the fact that a function contains control or data flow; thus, features encode additional relevant information such as the number of in- and out-degrees (ControlFlow) as well as operand size and p-code operation (DataFlow). In addition, features also encode behavioral information collected from neighboring features. For instance, `ControlFlow` features iteratively encode the features of incoming basic blocks as well as their edge type (true, false, fall through). In order to store and cluster binary code using LSH, BSim generates *numerical* representations

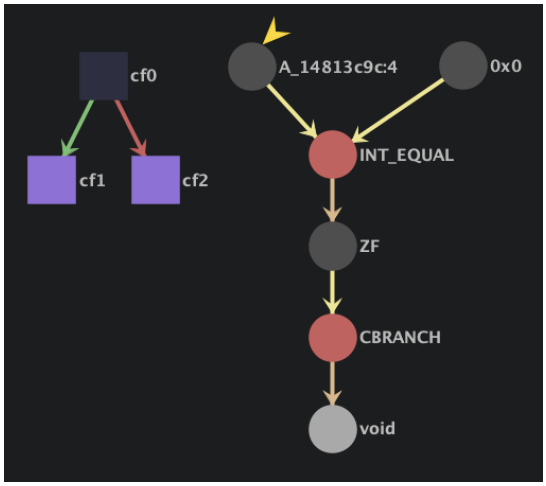

Figure 2: Feature Representation Example

for each distinct feature that retains this additional behavioral information.

The in-depth details of this procedure are out of scope for this paper, but it is sufficient to say that BSim 1) generates an initial 64-bit hash value for a feature rooted at either a basic block or a Varnode, then 2) iteratively combines the initial hash value with adjacent, related features. Both procedures rely heavily on bitwise operations, and the later uses a custom "hash mixin" algorithm that incorporates a precomputed, hard-coded Cyclic Redundancy Check (CRC) table. (Add a footnote here for the files where all of this is implemented).

2.1.1 DataFlow Features. Data flow features are generated from individual *Varnodes* in the p-code AST. *Varnodes* represent variable locations in the p-code IR and track their location (address), size (in bytes), defining p-code operation (in-degree in the AST), and all p-code operations that use it (out-degree in the AST). A `DataFlow` feature is initialized with an encoding of its:

- P-code opcode, and
- Storage location size

According to a user-facing configuration, a `DataFlow` feature may optionally encode:

- Either its offset in the address space, or
- The fact that it is a constant, and/or
- The fact that it is a global, and/or,
- The fact that it is an input

Additionally, *Varnodes* that are not written to in the function are not emitted as features and *Varnodes* that are copies of others are emitted as special `Copy` features.

Following initialization, each `DataFlow` feature is combined with the features generated for the inputs to its respective *Varnode*. This process effectively encodes the behavior of p-code expressions rather than individual opcodes or sub-expressions.

2.1.2 ControlFlow Features. Control flow features are similarly generated from individual *basic blocks* in the p-code CFG. A `ControlFlow` feature is first initialized with an encoding of its:

- In-degree, and
- Out-degree

Additionally, basic blocks that include a call instruction will result in another `ControlFlow` feature that encodes the type of the call - *direct or indirect*.

After initialization, each `ControlFlow` feature is combined with the features generated for its *incoming* basic blocks. If an incoming basic block ends in a conditional branch, the `ControlFlow` feature also encodes if the incoming edge is *True* or *False*.

2.1.3 Dual Flow and Combined Features. `DualFlow` and `Combined` features are a special set of features which incorporate *both* control and data flow information. In a *2-gram* pass, BSim iterates over overlapping pairs of p-code opcodes contained within each basic block in the function. If the first opcode is the *root* of an expression - `call` and `branch` variations, `store`, and `return` - then a special `DualFlow` feature is generated. Along with the `ControlFlow` feature for the basic block, this feature encodes, for both opcodes in the pair:

- The opcode, and
- Each `DataFlow` feature related to the opcode

Finally, for basic blocks that begin with one of the root opcodes, a special `Combined` feature will be generated that encodes the information listed above, but only for the *single* opcode.

In conjunction, these features encode the relationship between particular expressions and the basic blocks they are contained within. This is yet another normalization component that reduces the effects of common code structures as it is less likely that these relationships would change between slight variations in source or compilation settings.

Using BSim's visualization tool, we provide an example *combined* feature in Figure 2 which is generated from the code snippet in Figure 1. The control flow feature on the left is rooted at the first basic block in the snippet and encodes both the *True* and *False* edges from the conditional jump. The data flow feature on the right encodes two sequential p-code operations lifted from the code snippet, as well as their operand values and sizes. The first operation, `INT_EQUAL` is lifted from the `mov` and `test` instructions. Note that BSim encodes the size of the memory address (four bytes) in the feature. This operation produces a value for the zero flag (ZF) which is then used as the single operand of the `CBRANCH` operation lifted from the `jnz` instruction.

2.2 Comparison Metrics

With a high-level understanding of how BSim represents features of binary code, we next formalize its metrics for comparison - *similarity* and *confidence*. First, note that BSim performs comparison at a *function level*. Thus, each function is represented by a *feature vector* V_F of 64-bit hexadecimal values:

$$V_F = \{f_1, f_2, \ldots, f_n\} \tag{1}$$

2.2.1 Term Frequency and Inverse Document Frequency. In addition to the raw feature vector, BSim also stores a weight associated with each $f \in V_F$ stored in the database. These weights are determined according to the *Term Frequency (TF)* and *Inverse Document Frequency (IDF)* and are calculated during feature ingestion. Taking inspiration from the NLP community, a *term* is analogous to a particular feature $f \in V_F$; thus, the TF is the number of occurrences of a particular feature in a function. Following this logic, a *document* is analogous to the entire corpus of binary-level functions; thus the IDF is the inverse frequency of a particular feature across all functions.

Each TF and IDF are used as an index in a TF weights and IDF weights lookup table, respectively, which is provided in the configuration of a particular BSim database instance. This configuration is discussed in the following section, but note, for now, that the value at each index in the TF table provides a weight $1 \leq w_{TF}$ and, in the IDF table, $0 < w_{IDF} \leq 1$. In both tables, larger weights correspond to less frequent features. Thus, the coefficient for a particular feature $f \in V_F$ is:

$$f.coeff = \texttt{tf\_weights[f.tf]} \times \texttt{idf\_weights[f.idf]} \tag{2}$$

2.2.2 Similarity. The similarity between two feature vectors $V_F^{(1)}$ and $V_F^{(2)}$ is computed as their weighted *cosine similarity*. That is,

$$\texttt{Similarity} = \frac{\sum_{f \in V_F^{(1\cdot2)}} f.coeff}{\texttt{len}(\texttt{V}_\texttt{F}^{(1)}) \times \texttt{len}(\texttt{V}_\texttt{F}^{(2)})} \tag{3}$$

where $V_F^{(1\cdot2)}$ is the vector of shared features between $V_F^{(1)}$ and $V_F^{(2)}$ with *minimal* TF.

Given this formulation, similarity always provides a bounded value between 0 and 1 which should, in theory, be maximized for similar functions with unique features.

2.2.3 Confidence. The confidence metric is a less straightforward computation and involves a number of probabilities included in the configuration of the particular BSim database instance:

$$\sum_{f \in V_F^{(1\cdot2)}} f.coeff - \texttt{numflip} * \frac{\texttt{norm\_probflip0} + \texttt{norm\_probflip1}}{\max(\texttt{len}(\texttt{V}_\texttt{F}^{(1)}), \texttt{len}(\texttt{V}_\texttt{F}^{(2)}))} - \texttt{diff} * \frac{\texttt{norm\_probdiff0} + \texttt{norm\_probdiff1}}{\max(\texttt{len}(\texttt{V}_\texttt{F}^{(1)}), \texttt{len}(\texttt{V}_\texttt{F}^{(2)}))} + \texttt{addend} \tag{4}$$

where `numflip` is the number of hashes in the *shortest* vector not shared in the other and `diff` is the difference in length of $V_F^{(1)}$ and $V_F^{(2)}$.

Note the first term in the calculation is the numerator from the similarity calculation. The probabilities `norm_probflip0`, `norm_probflip1`, `norm_probdiff0`, and `norm_probdiff1` are calculated using a log normalization factor and base probabilities provided in the configuration. Again, refer to the following section for details regarding the configuration of BSim database instances.

As its name suggests, confidence is a supplementary metric for similarity and does not have an upper bound. In practice, we have found that confidence is often a clearer indicator of function similarity, and have encountered cases where similarity is less than 0.2 while confidence is 100.0 or higher.

```
<dbconfig>
 <info>
  <name>Medium 64-bit</name>
   <owner>Example Owner</owner>
  ...
   <major>0</major>
   <minor>0</minor>
   <settings>0x49</settings>
 </info>
 <k>17</k>
 <L>146</L>
 <weightsfile>lshweights_64.xml</weightsfile>
</dbconfig>
```

Figure 3: Configuration Template: medium_64.xml

2.3 Database Configuration

When instantiating a BSim database, it is required to provide a configuration template that determines the settings, weights, and probabilities used when generating, storing, and comparing binary code features. Configuration templates are simply XML-formatted files included within the BSim directory in a user's Ghidra installation. Figure 3 shows the configuration for the `medium_64` template, which is documented to be tuned for roughly one million (or less) 64-bit binaries. Note a few particular values; *Settings* is used as a bit mask for feature generation settings; value k and L are configurations for the underlying LSH database; and *weightsfile* refers to an additional file containing the TF and IDF weights tables mentioned in 2.2.1.

2.3.1 Feature Generation Settings. In 2.1, we refer to a number of values that may be optionally encoded as part of the feature generation process. The settings that determine the use of these options are toggled according to the *settings* value in the configuration template. From our analyses of BSim's source code, the options are as follows:

- Truncate Varnode Size to Four Bytes, 0x1
- Remove Indirect Varnode Copies, 0x2
- Terminate Dataflow at Call Sites, 0x8
- Do not use Constant Value, 0x10
- Do not use Fact that Varnode is an Input, 0x20
- Do not use Fact that Varnode is a Global, 0x40

In practice, we find that the options to terminate dataflow at call sites and not use the fact that a Varnode is an input are not effectual as the first is never referenced during feature generation and checks for the latter are commented out in the source code. We also find that BSim performs a right bit shift of length two before loading settings, which enables a check bit to be appended to the bit mask for validation.

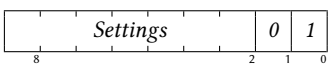

Figure 4: Feature Generation Settings Bit Mask

Thus, the settings value 0x49 in Figure 3 actually corresponds to a settings value of 0x12, which enable the options to not encode constant values and remove indirect copies of Varnodes.

2.3.2 Probabilities and Weights. The file referenced by the *weightsfile* entry in Figure 3 defines three primary tables; one each for coefficients relating to weighted probabilities of term and inverse document frequency and a third serving as a lookup table for feature hashes and their corresponding IDF. An example taken from our Ghidra installation is shown in Figure 5.

```
<weights settings="0x49"> <!-- Created 11/14/23 -->
<weightfactory scale="1.51275976" addend="6.25597601">
 <idf>1.00000000e+00</idf>
 <idf>9.99459306e-01</idf>
 ...
 <idf>6.68999073e-01</idf>
 <tf>1.00000000e+00</tf>
 ...
 <tf>2.64145413e+00</tf>
 <tf>2.64575131e+00</tf>
 <weightnorm>1.35049281e+01</weightnorm>
 <probflip0>2.02671876e-01</probflip0>
 <probflip1>5.40692533e-01</probflip1>
 <probdiff0>5.19701356e-02</probdiff0>
 <probdiff1>8.52635318e-01</probdiff1>
</weightfactory>
<idflookup size="1000">
<hash count="0">0x5448c6df</hash>
<hash count="0">0x5e3fe72a</hash>
<hash count="0">0x8732d39a</hash>
<hash count="0">0xc530e221</hash>
<hash count="1">0x15231688</hash>
<hash count="1">0x4af9a820</hash>
...
<hash count="508">0xd5574099</hash>
<hash count="509">0x52f765fa</hash>
<hash count="510">0xc55041c4</hash>
<hash count="511">0xab6831d3</hash>
</idflookup>
</weights>
```

Figure 5: Weights File: lshweights_64.xml

The two weights tables are included in an outer *weightfactory* entry that is loaded by BSim into a respective object at runtime. As stated in 2.2.1, the weights in the IDF table range between $0.0 < w \leq 1.0$. This table contains 512 unique entries and, as seen in the *idflookup* table in Figure 5, is indexed by looking up a "count" for a particular feature. However, the *idflookup* table contains *1000* entries with some count values duplicated. From our analysis, we conclude that the *idflookup* table contains the 1000 most frequent features, according to BSim's tuning, and features not included in this table are given a default count value 0 and, thus, an IDF weight of 1.0. Also stated in 2.2.1, the TF weights table is indexed using a feature's frequency within a single function. This table contains 64

Table 1: Experiment Databases

Language	Compiler	Package	Opt. Level	Functions
C	Clang	GNU Coreutils	O0	17,471
C	Clang	GNU Coreutils	O1	11,981
C	Clang	GNU Coreutils	O2	12,380
C	Clang	GNU Coreutils	O3	11,659
Rust	Rustc	Uutils Coreutils	O0	563,638
Rust	Rustc	Uutils Coreutils	O1	147,420
Rust	Rustc	Uutils Coreutils	O2	128,553
Rust	Rustc	Uutils Coreutils	O3	113,144
Go	gc	Homebrew + Arch User Utils	Default	605,665
Go	gc	Homebrew + Arch User Utils	NoInline (-l)	606,396
Go	gc	Homebrew + Arch User Utils	NoOptimization (-N)	605,688
Go	gc	Homebrew + Arch User Utils	Asan (-asan)	605,955
Go	gc	Homebrew + Arch User Utils	Msan (-msan)	605,899

unique values, and, for any feature with a frequency greater than 64, the default weight is the last entry in the table.

The additional probabilities and scaling factors discussed in Section 2.2.3 are also shown in Figure 5. Note the normalized probabilities `norm_probflip0`, `norm_probflip1`, `norm_probdiff0`, and `norm_probdiff1` are calculated by multiplying the *scale* value with the respective base probabilities. Our analysis also finds no reference to the *weightnorm* entry that affects the generation or comparison of features.

3 RESEARCH QUESTIONS

For the remainder of this paper, we utilize BSim as a state-of-the-art binary similarity tool to address a number of important research questions. Primarily, we investigate how source language affects the efficacy of binary similarity techniques in terms of overall overall accuracy; though, we also provide results for adjacent questions concerning the particular comparison mechanism and feature representations. Our three research questions are as follows:

- **RQ1.** Does source language degrade binary similarity, and to what extent?
- **RQ2.** How does source language affect clustering-based binary similarity tools?
- **RQ3.** Can features be tuned according to their source language?

Addressing these questions is important for the immediate improvement of BSim and similar tools, but also provides other research directions in the development of more flexible binary similarity solutions. Further, their answers serve as guidance for binary analysis practitioners when applying existing tools to real-world problems.

4 EVALUATION

In this section, we address and provide results for each of the research questions defined in 3. For each study, we include both a baseline result from a C-based dataset and an experimental result from a Rust-based dataset. This work is inspired by a desire to improve the landscape of binary analysis tools for the Rust language, though we could expand this study to other compiled languages, such as Golang, in future work.

Table 2: Overall Accuracy - Baseline (C)

Query Level, DB Level	Accuracy (%)
O0,O0	100.0
O0,O1	69.20
O0,O2	67.79
O0,O3	66.60
O1,O1	100.00
O1,O2	98.83
O1,O3	97.80
O2, O2	100.00
O2,O3	99.26
O3,O3	100.00

4.1 Datasets and Setup

For our baseline C-based dataset, we use a subset of binaries provided by the BinKit work [11]. Specifically, we include binaries from *GNU Coreutils 9.1* compiled with *Clang 13.0* for the *x86_64* architecture at optimization levels *O0, O1, O2, and O3*. This dataset includes a total of 400 binaries and 53,491 functions.

Our experimental Rust-based dataset is built from the **uutils coreutils** project, which "aims to be a drop-in replacement" for the GNU Coreutils [18]. We similarly compile each utility using the standard Rust compiler, *rustc 1.75* for the *x86_64* architecture at optimization levels *O0, O1, O2, and O3*. This dataset also includes a total of 400 binaries with 952,755 functions. The significant difference in the number of functions is due to 1) static linking of Rust dependencies and 2) high-level constructs like closures and default trait implementations that are not used in C. These factors are not anomalous to our dataset and are standard behavior for Rust binaries. Additionally, BSim does not allow function-level ingestion or removal of single functions from a database. Thus, we include the entirety of each binary in our experiments.

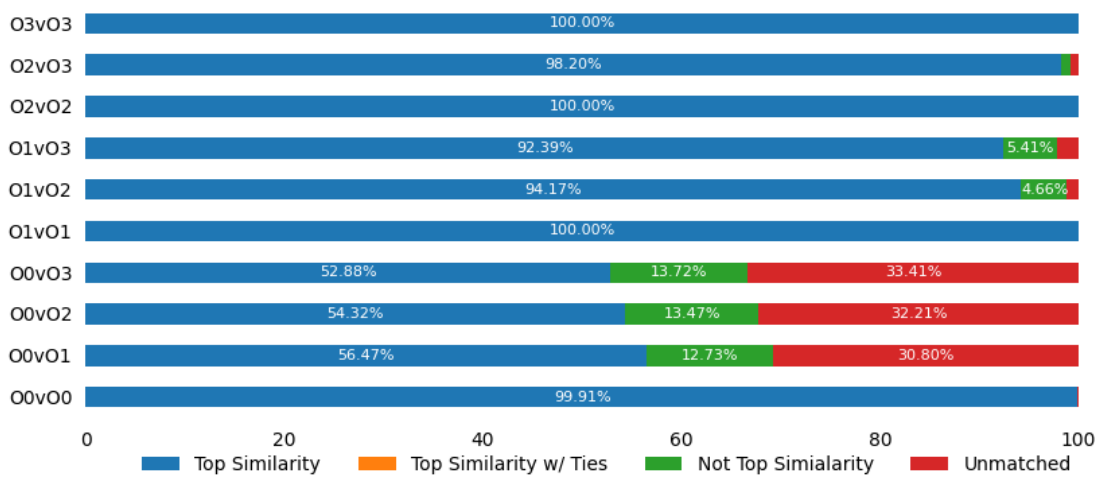

Figure 6: Top Match Accuracy - Baseline (C)

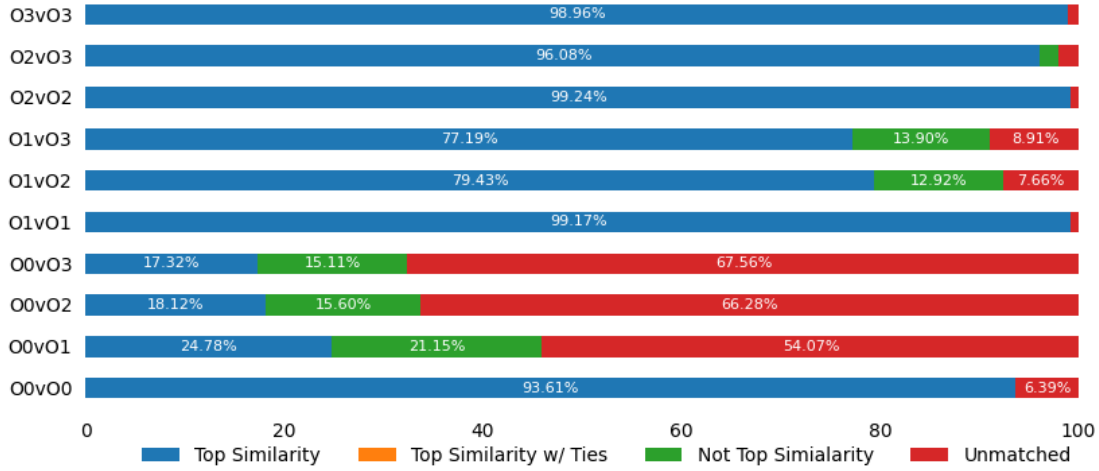

Figure 7: Top Match Accuracy - Experimental (Rust)

In both databases, we compile with all debug information in order to retain symbols for our main source of ground truth.

To setup our experiments, we sought to create as realistic of an experimental environment as possible, reflecting that of a real-world binary similarity workflow. As such, we created one BSim database for each distinct compiler configuration in both datasets. That is, we created one database for each optimization level, holding the compiler and the source language constant. Each binary was analyzed and ingested into its respective database using Ghidra 11.2. The contents of our databases are summarized in Table 1.

In conducting our experiments, we executed and recorded queries for each unique permutation of compiler settings, again holding the source language and compiler constant. For each permutation, we use the first setting component as the *source of queries* and the second as the database *to be queried*. For example, with the permutation {`coreutils_clang_O0`,`coreutils_clang_O1`}, we perform a query for each function in the O0 set on the O1 database. We configure BSim to report, at most, the 1000 nearest neighbors of a function until the correct symbol is returned. Along with the symbol names from the queries, we also record the reported similarity and confidence for each result.

Towards **RQ3**, we also collected the features generated for each function in both datasets, as well as their respective TF and IDF.

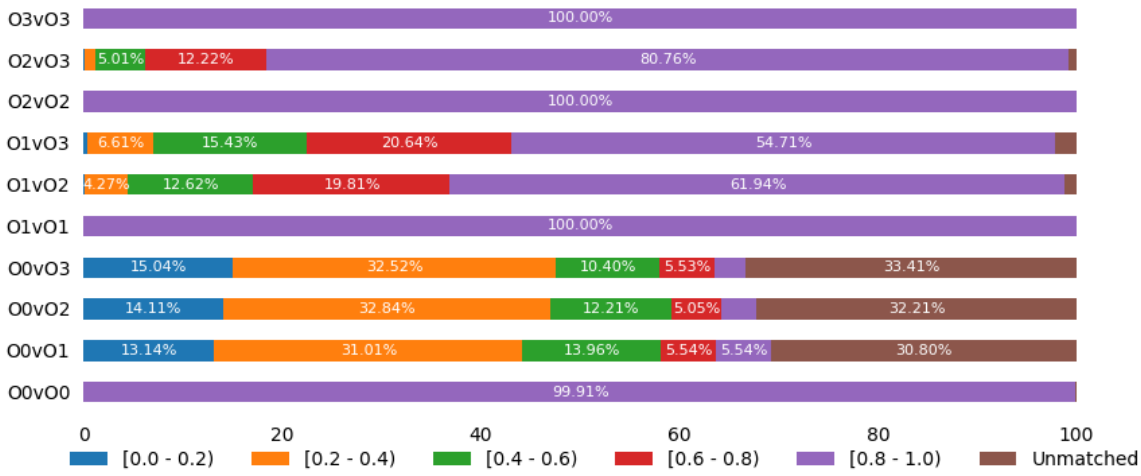

Figure 8: Match Accuracy by Similarity - Baseline (C)

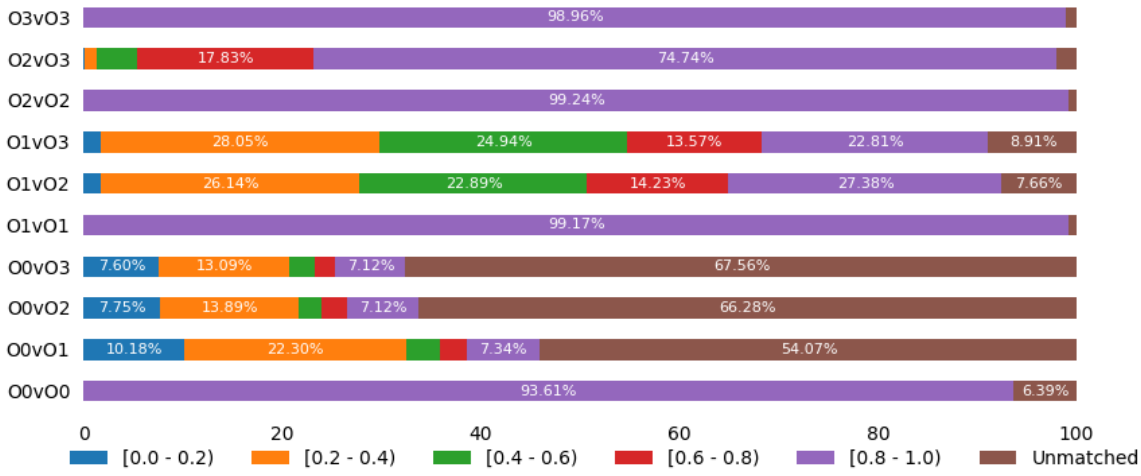

Figure 9: Match Accuracy by Similarity - Experimental (Rust)

4.2 Results

With our collected data, we conducted a number of analyses towards answering the research questions presented in Section 3. In this section, we provide the results of these analyses for both the baseline and experimental cases. In addition, we include discussion for each particular analysis and how it satisfies in answering our posed questions.

4.2.1 RQ1. The most fundamental of our research questions, **RQ1** questions the existence of an effect on binary similarity due to the source language. For this, we report the overall accuracy of BSim, where a successful trial is recorded if a candidate function symbol matches the queried function symbol, at *any* similarity level. Thus, for each row in Tables 2 and 3, we report the rate at which BSim returned the correct symbol within the 1000 nearest neighbors of each queried function.

Given these results, we find that the source language does, in fact, affect binary similarity tools, even in a state-of-the-art solution like BSim. Interestingly, we find that a number of functions were not successfully matched when querying the database generated from the same binary (the O0vO0, O1vO1, O2vO2, and O3vO3 rows). This is to say that there were 1000 *other* functions that BSim found to be more similar than the function itself.

It is worth noting that the most drastic difference between the baseline C results and the experimental Rust results were when

Table 3: Overall Accuracy - Experimental (Rust)

Query Level, DB Level	Accuracy (%)
O0,O0	93.61
O0,O1	45.93
O0,O2	33.72
O0,O3	32.43
O1,O1	99.17
O1,O2	92.35
O1,O3	91.09
O2, O2	99.24
O2,O3	97.90
O3,O3	98.96

comparing an unoptimized (O0) binary with a database populated by an optimized binary. This is an important distinction, given that, in most real-world settings, binaries are not distributed with optimizations. Thus, practitioners should be cognizant if populating databases from binaries they have compiled for tasks like library detection. Nonetheless, even when comparing binaries with optimizations enabled, our experimental results indicate a degree of degraded efficacy.

4.2.2 RQ2. In the previous section, we indicated that BSim performs approximate nearest neighbor queries, returning not a singular match for a given function, but a configurable set of probable similar functions. This technique has proven to alleviate some of the issues faced when comparing binaries of varying compilation settings and searching across a large corpus. However, even when expanding the range of approximation to 1000 results per function, the results for **RQ1** demonstrated a degraded performance due to variation in the source language. Thus, we use **RQ2** to investigate the ability of BSim to determine the correct function with a maximum similarity across all matches.

Figures 6 and 7 extract additional information from our accuracy results, now partitioning the frequency of correct matches into either Top Similarity or *Not* Top Similarity. Note, there is also an entry in the legend for results which *tied* for the top match by similarity, though we did not experience this in either the baseline or experimental datasets.

In the partition for *Not* Top Similarity we find that frequencies double or nearly double between the C and Rust experiments. So, not only does the source language degrade the overall performance of BSim, it also degrades the rate at which the top result (by similarity) is actually the correct result. This does not suggest, however, that clustering techniques are not effective for binary similarity, but does suggest that the feature representation and/or tuning may be skewed for binaries produced by C.

We further partition the results into categories by range of similarity in Figures 8 and 9. The baseline results show that, in the majority, correct matches for C tend to be of higher similarity than those for Rust. With the exception of the unoptimized comparisons (O0 v. O1, O2, O3), the experimental results show that correct matches for Rust occur in lower similarity ranges more often. Most interesting is the result for the O1 comparisons; notice that, despite the comparatively similar *overall* accuracy, the similarity scores have far more variability for Rust than for C. In a practical sense, practitioners may receive correct matches at a comparable rate with no query parameters for similarity or number of matches; but, if filtering for matches within only a certain threshold, their results may be severely degraded.

Table 4: Feature Overlap

Language	Overlap	Occurrence (Avg.)
C	876	1365.66
Rust	737	360.53

4.2.3 RQ3. Our final research question is motivated by the results presented above and, as a Work in Progress paper, this version does not include a full conclusion for **RQ3**. Primarily, this question seeks to provide an immediate solution for the degradation of efficacy experienced by BSim when comparing binaries produced by Rust. BSim exposes a convenient interface for tuning features according to their frequency in the corpus - as described in Section 2 - and thus provides a simple direction for continuing this work. However, as we have stated, there exists no clear documentation or tooling to *generate* the tuning schemes used by BSim, so we cannot fully address this question. Nonetheless, we present a few early results which necessitate further investigation.

Recall from Section 2.3.1 that the BSim weights configuration file contains the 1000 most frequent features, which are then used to index a table of weight coefficients in order to calculate similarity and confidence. Using the features collected from both datasets, we cross-reference those seen in our experiments with the top 1000 features included in the `lshweights_64.xml` file and record the number of times they were seen.

The *Overlap* in Table 4 records the number of features from the 1000 most frequent in the configuration file that occurred throughout each dataset. The *Occurrence* values are an average of the number of times *each* feature was seen in its respective dataset. Thus, the table summarizes an interesting finding; not only are the features in the configuration file seen fewer times in the Rust dataset than the C dataset, but they are drastically less representative of the features seen in Rust binaries.

5 DISCUSSION AND FUTURE WORK

Our results for **RQ1** and **RQ2** clearly indicate that variation in the source language negatively affects the efficacy and performance of binary similarity tools. Further, our assessments demonstrate that the quality of results in probability- and clustering-based similarity tools are degraded when binaries are generated from non-C source languages. Finally, our initial investigation into **RQ3** provides important insight into the necessary direction of future work.

5.1 Feature Weight Tuning

This work clearly necessitates the investigation of further tuning, either for broader generality or increased specificity for particular languages, applications, and/or domains. With its accessible interface for configuration and tuning, BSim is uniquely positioned to address this need. However, without complete documentation and/or tooling, we are currently unable to determine a process that

derives the weights configuration file described in Section 2. An open discussion post in the Ghidra GitHub repository confirms our findings that the source code used to derive and tune weights is not distributed. The inquiry has since been elevated to the Ghidra development team. As we present this ongoing work, we leave the open-sourcing and/or documentation of the weight tuning mechanism to future work with *highest* importance. In doing so, BSim will be fully equipped to investigate the full extent of our research questions, along with many others that may find motivation from this work.

5.2 Improving Analyses

Also included in Section 2, we note that BSim utilizes *p-code*, Ghidra's intermediate representation (IR), which is lifted from a binary's raw machine code. As some early work has documented, the quality of this process is complicated by non-C languages, like Rust, and may be an additional source of limitation in generating quality features for our Rust dataset. From this perspective, we are interested in pursuing analysis procedures for generating higher quality IR from Rust binaries, such that feature generation is able to produce richer information. For example, recent work has utilized pointer analysis to derive more accurate call graphs, which may be beneficial for determining more precise data flow across call boundaries [12]. Other static analysis techniques may be useful for determining function arguments, which are typically a source of incompleteness when analyzing Rust binaries [9].

6 CONCLUSION

Efficient and accurate binary code similarity is a complex, yet vital technique for real-world binary analysis. While the area has seen sustained research effort to improve and develop techniques, binary similarity is complicated by a number of factors including compiler versions, optimization settings, and now, as we have demonstrated, source code language. Using a state-of-the-art tool, BSim, this work shows that the source language does, in fact, affect the quality of binary similarity results. Our assessments suggest that current tooling is unable to match binary code generated from the Rust language with reliable accuracy and necessitates further attention. We present two main research directions to address the issues found in this work through additional tuning of language-specific features and improved IR lifting procedures. We also provide an introduction of BSim to the open research record and hope to see its further investigation and adoption in future work.

REFERENCES

[1] National Security Agency. 2023. BSim Tutorial. Retrieved December 2, 2024 from https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraDocs/GhidraClass/BSim/README.md.

[2] Ulrich Bayer, Paolo Comparetti Milani, Hlauschek Clemens, Christopher Kruegel, and Engin Kirda. 2009. Scalable, Behavior-Based Malware Clustering. In *Network and Distributed System Security Symposium (NDSS)*.

[3] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. 472–489. https://doi.org/10.1109/SP.2019.00003

[4] Thomas Dullien. 2004. Structural Comparison of Executable Objects. *DIMVA* (07 2004). https://doi.org/10.17877/DE290R-2007

[5] Thomas Dullien and Rolf Rolles. 2005. Graph-based comparison of executable objects (english version). *SSTIC* 5 (01 2005).

[6] Mohammad Reza Farhadi, Benjamin C. M. Fung, Philippe Charland, and Mourad Debbabi. 2014. BinClone: Detecting Code Clones in Malware. In *Proceedings of the 2014 Eighth International Conference on Software Security and Reliability (SERE '14)*. IEEE Computer Society, USA, 78–87. https://doi.org/10.1109/SERE.2014.21

[7] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Information and Communications Security*, Liqun Chen, Mark D. Ryan, and Guilin Wang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 238–255.

[8] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 896–899. https://doi.org/10.1145/3238147.3240480

[9] Ben Herzog. 2023. Rust Binary Analysis, Feature by Feature. Retrieved December 2, 2024 from https://research.checkpoint.com/2023/rust-binary-analysis-feature-by-feature/.

[10] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. 2009. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '09)*. Association for Computing Machinery, New York, NY, USA, 611–620. https://doi.org/10.1145/1653662.1653736

[11] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2023. Revisiting Binary Code Similarity Analysis Using Interpretable Feature Engineering and Lessons Learned. *IEEE Transactions on Software Engineering* 49, 4 (April 2023), 1661–1682. https://doi.org/10.1109/tse.2022.3187689

[12] Wei Li, Dongjie He, Yujiang Gui, Wenguang Chen, and Jingling Xue. 2024. A Context-Sensitive Pointer Analysis Framework for Rust and Its Application to Call Graph Construction. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction* (Edinburgh, United Kingdom) *(CC 2024)*. Association for Computing Machinery, New York, NY, USA, 60–72. https://doi.org/10.1145/3640537.3641574

[13] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1157–1177. https://doi.org/10.1109/TSE.2017.2655046

[14] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How Machine Learning Is Solving the Binary Function Similarity Problem. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2099–2116. https://www.usenix.org/conference/usenixsecurity22/presentation/marcelli

[15] Derrick McKee, Nathan Burow, and Mathias Payer. 2023. Accurate Compiler and Optimization Independent Function Identification Using Program State Transformations. https://doi.org/10.14722/bar.2023.23003

[16] Brian Ruttenberg, Craig Miles, Lee Kellogg, Vivek Notani, Michael Howard, Charles LeDoux, Arun Lakhotia, and Avi Pfeffer. 2014. Identifying Shared Software Components to Support Malware Forensics. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Sven Dietrich (Ed.). Springer International Publishing, Cham, 21–40.

[17] Zeming Tai, Hironori Washizaki, Yoshiaki Fukazawa, Yurie Fujimatsu, and Jun Kanai. 2020. Binary Similarity Analysis for Vulnerability Detection. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. 1121–1122. https://doi.org/10.1109/COMPSAC48688.2020.0-110

[18] uutils. 2024. coreutils. Retrieved December 2, 2024 from https://github.com/uutils/coreutils.

[19] Hao Wang, Zeyu Gao, Chao Zhang, Mingyang Sun, Yuchen Zhou, Han Qiu, and Xi Xiao. 2024. CEBin: A Cost-Effective Framework for Large-Scale Binary Code Similarity Detection. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 149–161. https://doi.org/10.1145/3650212.3652117

[20] Shouguo Yang, Zhengzi Xu, Yang Xiao, Zhe Lang, Wei Tang, Yang Liu, Zhiqiang Shi, Hong Li, and Limin Sun. 2023. Towards Practical Binary Code Similarity Detection: Vulnerability Verification via Patch Semantic Analysis. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 158 (Sept. 2023), 29 pages. https://doi.org/10.1145/3604608

Received 13 December 2025