*mathematics*

MDPI

*Article*

# Assembly Function Recognition in Embedded Systems as an Optimization Problem

**Matan Avitan [1], Elena V. Ravve [2,*] and Zeev Volkovich [2]**

[1] Rafael, Advanced Defense Systems Ltd., Haifa 3102102, Israel
[2] Department of Software Engineering, Braude College of Engineering, Karmiel 21982, Israel; vlvolkov@braude.ac.il
* Correspondence: cselena@braude.ac.il

**Abstract:** Many different aspects of software system development and verification rely on precise function identification in binary code. Recognition of the source Assembly functions in embedded systems is one of the fundamental challenges in binary program analysis. While numerous approaches assume that the functions are given a priori, correct identification of the functions in binaries remains a great issue. This contribution addresses the problem of uncertainty in binary code in identification of functions, which were optimized during compilation. This paper investigates the difference between debug and optimized functions via modeling of these functions. To do so, we introduce an extensible model-centred hands-on approach for examining similarities between binary functions. The main idea is to model each function using a set of predetermined, experimentally discovered features, and then find a suitable weight vector that could give impact factor to each such a feature. After finding the weight vector, the introduced models of such desired functions can be identified in binary software packages. It means that we reduce the similarity identification problem of the models to a classical version of optimization problems with one optimization criterion. Using our implementation, we found that the proposed approach works smoothly for functions, which contain at least ten Assembly instructions. Our tool guarantees success at a very high level.

**Keywords:** optimization; debug compilation; optimizing compilation; assembly function; function recognition; embedded systems

**MSC:** 90C47; 49K35; 90C90; 93-04

check for updates

**Citation:** Avitan, M.; Ravve, E.V.; Volkovich, Z. Assembly Function Recognition in Embedded Systems as an Optimization Problem. *Mathematics* **2024**, *12*, 658. https://doi.org/10.3390/math12050658

Academic Editor: Andrea Scozzari

Received: 23 September 2023
Revised: 30 October 2023
Accepted: 22 December 2023
Published: 23 February 2024

**Copyright:** © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

## 1. Introduction

Nowadays, lots of third-party libraries may be down-loaded from open source repositories and reused. In [1], it was reported that *"7% to 23% of the code in a typical software system has been cloned"*. In fact, in lots of modern C++ libraries, we might find a lot of functions, which are identical byte-to-byte while have different names. A profound understanding of why software developers reuse code may be found in particular in [2].

In this case, one of the major challenges in reverse engineering is understanding the real purpose of programs written in modern languages at a high level. This may be considered useless because no ability to spend the time analyzing external programs and significant algorithms that the manufacturer has attached to its main programs. Unfortunately, this process should be repeated on the same directory functions time after time in reverse engineering. Sometimes, the more profound understanding of a class of library function can significantly improve program analysis. This understanding may be very useful in the similarity analyzes.

In fact, there are recent attempts to automatically generate code comments for the functional units, cf. [3], that may help developers to maintain programs. However, such comments are often omitted or archaic in software systems. It means that software engineers must extract the functionality from the source (binary) code.

Function recognition is an example of a basic backend engineering challenge in binary program analysis. Also, many expressions of this type of challenge can be found in an open source libraries. Many manufacturers use open source libraries' code to develop their products and this is one of the most prominent principles in software engineering called reuse. Open source code is embedded in their products as one or more binary files or as part of the firmware.

When performing analysis on those binaries, it is possible to find the original Assembly code, which runs in the manufacturer's product. Usually, the full firmware is divided into several sections of a code, called a function, which requires easier and faster code analysis. It makes identification of library functions very easy without having to read them and without spend time on this. Standard library functions or just *standard functions* are sometimes more than 90% of all functions of a software system.

For instance, as it is shown on Figure 1, one well known compiled the "Hello, world!" program (see Figure 2) contains: library functions–294; function main()–1. Of course, this is an artificial example. However, this particular case shows that the main function contains an average of 50% of open source libraries. Therefore, the researcher is forced to spend lot of time analyzing the library functions. During such a study, meaningful comments and names in the function can provide a faster understanding of the purpose.

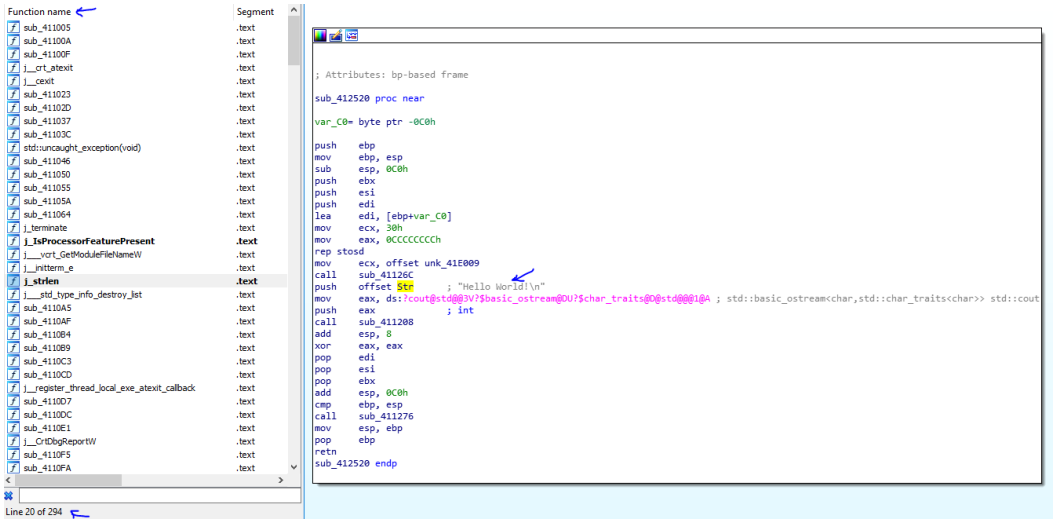



**Figure 1.** Compilation of the program.

```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}
```

**Figure 2.** Initial program.

In fact, there is no intention to achieve a perfect identity, it is impossible in 100% of cases even theoretically. Moreover, recognizing certain functions can lead to unwelcome consecutions. In fact, the "unmasking" of thenext function:

```
1.  push  bp
2.  mov   bp,sp
3.  xor   ax,ax
4.  pop   bp
5.  ret
```

would lead to many misidentifications.

As it is shown on Figure 3, after compiling an entire library (e.g., libc.so) and creating a binary file, one wants to use it to identify all functions in the binary code, provided by

the manufacturer. Inspired by the examination, in this paper, we propose a possible answer to the following question, see Figure 4:

> *How to adjust between the Assembly optimization function taken from a product and the Assembly function, which is known to compile in a debug environment that was downloaded from an open source repository independent architecture?*

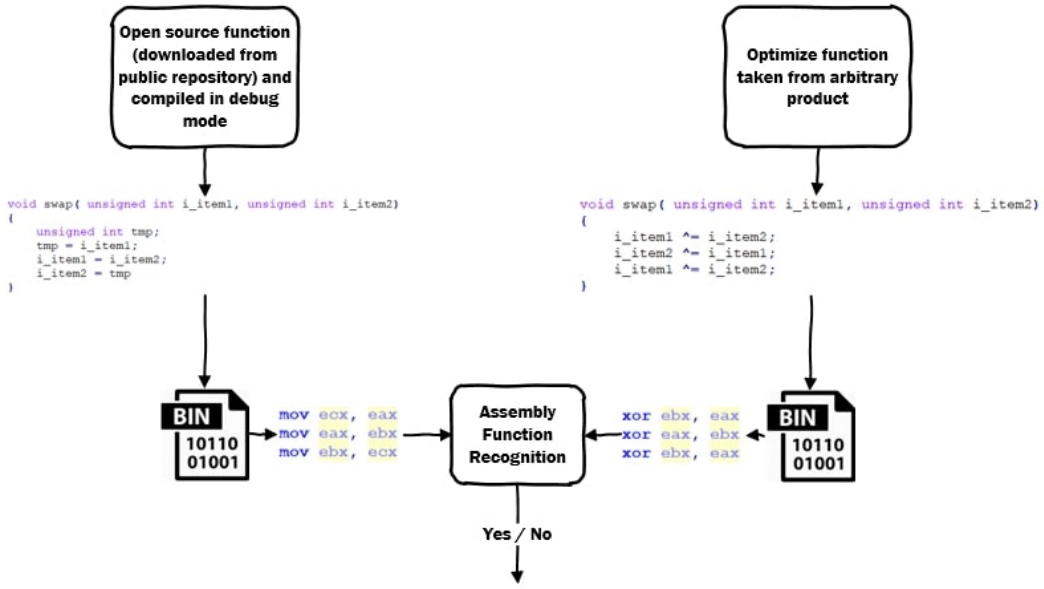



**Figure 3.** Recognition of differences in programs.

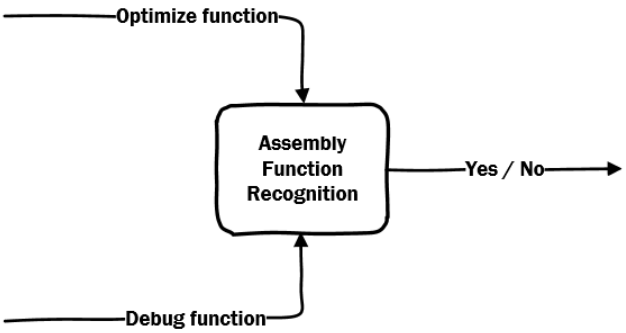



**Figure 4.** The main question to be answered.

To do so, we compare models of the functions, which we introduce in the paper, rather than the functions themselves. In this approach, we follow [4,5], which in particular claim that *"Models and modelling is the fourth dimension of Computer Science!"*

**Definition 1** (Generic notion of a model, taken verbatim from [5])**.** *A model is a well-formed, adequate, and dependable instrument that represents "something" (called origin as a source, archetype, starting point) and functions in scenaria of use.*

In fact, we score each function using a set of predetermined, experimentally discovered features and then find a suitable weight vector that could give impact factor to each such a feature to the similarity between pairs of functions. In this way, we convert the similarity identification problem to a classical version of optimization problems with one optimization criterion. The set of the features may be revised and extended by using more a posteriori data. The general approach is rather illustrated on Figure 5.

The proposed approach leaves some open questions as well as other directions of research:

- In order to evaluate quality features, used for the similarity definition, we used a *touchstones*, see Section 4. The question that arises immediately is: do the results, shown on Figures 17–21, depend upon the chosen set of *touchstones*? If so, what should be the *golden set* of the *touchstones* if any?
- In Section 5, we converted the similarity identification problem to a classical version of optimization problems with one optimization criterion. Another option is to use some version of multi-criteria optimization.
- Using our PoC (Proof of Concept) implementation, we found that the proposed approach works smoothly for functions, which contain at least ten Assembly instructions. We encountered difficulty in identifying functions, which contain a small number of instructions. In fact, this is not surprising: as small the number of instructions per a function as less any difference may be informative (Consider two functions each containing only one assignment or only one conditional jump. How would you catch a difference between them?!). The observation erases questions such as what is the minimum number of Assembly instructions per a function that the proposed approach may handle? Is it possible to add a feature that solves the lack of information problem in small functions instead of solving the lower block problem?
- In Section 5.2, we took `uClibc`, which demonstrated 93% of success. It would be interesting to investigate additional library functions with the same features.
- A study of adding new meaningful features to Assembly code would only improve the approach for better results and accuracy.

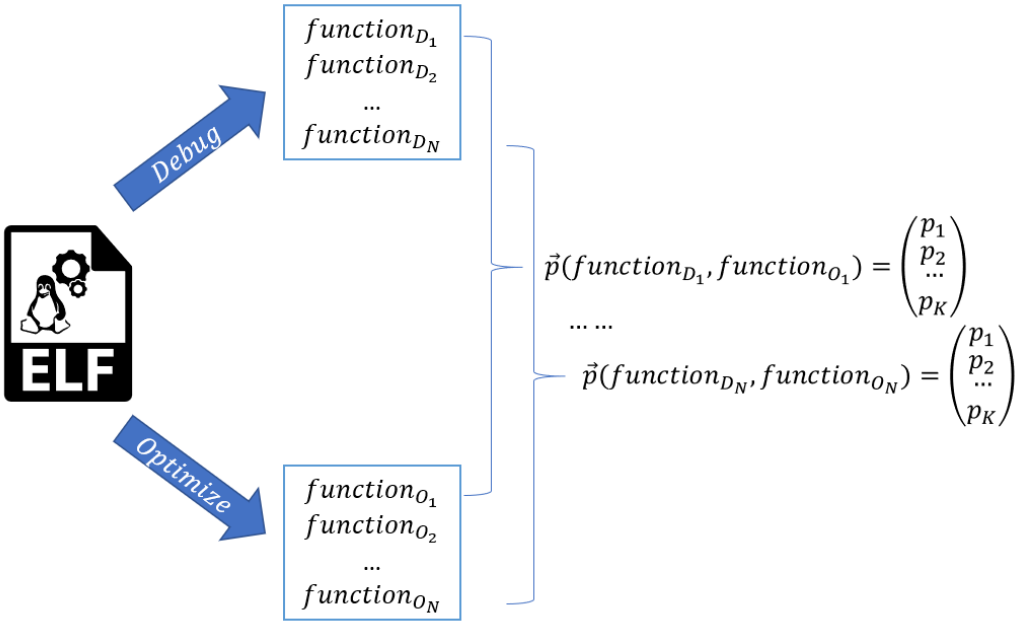



**Figure 5.** The general approach.

The rest of the paper is organized in the following way. In Section 2, we present some approaches, which are the most relevant to our contribution. In Section 3, we propose five features aimed to answer our main question of Figure 4. In Section 4, we introduce a statistical approach, that allows estimation of the quality of the proposed features. We show that at least two of the proposed features differ significantly according to the approach. We conclude that the proposed features might not reflect in fact similarity of codes. In Section 5, we introduce an optimization approach to a definition of the impact of every chosen feature on the final (binary: *yes* or *no*) decision if the pieces of codes under consideration in fact are similar. In Section 6, we provide PoC (Proof of Concept) implementation of our approach. Section 7 summarises the paper and provides outlook.

## 2. Related Work

In this section, we recall some approaches, which are the most relevant to our scope of interest. We explicitly declare that in this section some sentences are taken almost verbatim from the corresponding sources.

### *2.1. Similarity Measures for Source Code*

Numerous works on detecting similarities in source code may be mentioned, starting from [6]. Abstraction of *Program Dependence Graphs* (PDG), derived from functions using their source code, may be one of the first successful attempts, cf. [7]. It opened a door for hundreds of contributions. Here, we report only the most recent and the most relevant of them: [8–13]. In addition, the abstraction may be combined with machine learning methods as shown in particular in [14]. However, most of them are not applicable to the case of binary. In particular, Assembly code has very limited information about the used variables (like, types), cf. [15]. The observation immediately limits the use of PDG for our purposes.

### *2.2. Similarity in Binaries*

As it was shown in [16], iterative compilation as well as machine learning-based optimization machinery have been affirmed to be extremely successful in reaching optimizations. In [17], one finds an approach, which operates in a similar context of recognition of code-reuse as well as IP piracy in binary code, using symbolic execution and theorem provers. The authors miss a cross-compiler evaluation and adopt some syntactic heuristics in form of number of arguments which may significantly affect presiciness.

The approach, proposed in [18] is also thickly connected to syntax. That is why it hardly handles differences, caused by the use of different compilers, as shown by [19].

### *2.3. Similarity of Binaries through Re-Optimization*

Authors of [20] present a scalable approach for defining similarity between so-called *stripped* binary codes. In this case, no debug information is assumed. The authors are picking with the use of different compilers, different optimization options as well as different architectures. The presented technique is scalable and precise due to "out-of-context re-optimization of procedure fragments". Each binary procedure is decomposed to comparable fragments, which are free of compiler optimizations. The "cleaning" allows finding equivalent fragments by using straight-forward syntactic comparison. In particular, the approach inspired a recent efficient CNN-based compiler optimization recognition system as shown in [21].

### *2.4. Dynamic Approaches*

The dynamic approach of [22] is based on comparison of results of procedures, which are executed using random values. The dynamic approach of [23] is based on comparison of the traces of the executions rather than on the results. In general, the approach could be applied to our binary similarity hunting. Unfortunately, it can be hardly scalable. Moreover, dynamic approaches are pretty limited due to significant difference between executions and traces on different hardware. However, it was recently used in [24,25].

### *2.5. Structure-Based Static Methods*

The authors of [26] present an impressive approach for finding similarity in software using similarity of the corresponding expression trees. However, any code motion might affect the correctness of the reported results. That is why it is not suitable for our case since different compilers produce different "layouts" of code for the same computation.

In [27], an approach to identify binary procedures by the sequence of the involved system calls is presented. This method is only applicable to procedures with no indirect calls or use of direct system calls. Authors of [28] investigate different architectures. They consider the *"static control flow structure"* of procedures in order to detect their similarity. In order to guarantee scalability of the method, they use a numeric filtering pre-stage.

However, the proposed solutions, based mostly on syntactic features might affect accuracy, in particular in the cases of procedures with a small number of branches.

Horwitz et al., cf. [7], on the one hand, recognize the importance of statistical reasoning. Yet, their method is strongly tailored to source code, which makes it hardly applicable to our case of binary codes similarity. Jang et al., cf. [29], use a very close $n$-gram decomposition, for malware detection using features. In addition, another $n$-grams based approach is presented in [30], in combination with graphlets. However, it suffers from the same issues as all similar approaches described above.

### *2.6. Equivalence Checking and Semantic Differencing*

Partush et al., cf. [31,32], examine `C` programs with loops and develop an abstract vision of program differences for them. The approach can not be adopted to our particular case as it does not operate on binaries.

In [33], `SymDiff` uses a program verifier to prove equivalence of procedures. In this case, the procedures are translated to `BoogieIVL` and either reported to be similar or a counter example is provided. Unfortunately, the approach is pretty limited for the case of loops. Moreover, it requires the translation of the full software system to the binary. Furthermore, the use of provers makes the approach to be hardly scalable.

Engler et al., cf. [26], are hunting for bugs in different implementations of the same procedures. To reach the aim, they present a symbolic approach for detecting procedure equivalence. The method does not work in our case as, on the one hand, it considers the source code rather than binary and, on the other hand, the method is limited to non-looping code, that is absolutely not acceptable for us.

In [33], the authors are concentrated on the equivalence of recursive functions and use a recursion rule to detect this particular similarity. Again, the rule operates at the source code, but not binary. Moreover, the used theorem prover limits its scalability. Once more, contributions of [27,31,34] are dealing with (versions of) procedures in high-level code, but not binary.

The authors of [31,34], on the one hand, handle loops. However, they use very expensive from the computational point of view libraries, which is again not acceptable for us. Sharma et al., cf. [23] introduce machinery for detecting equivalence between high-level code and binary even with loops. The machinery is based on a simulation relation between the high-level code and the binary. However, again, the implementation is very complicated from the computational point of view and it is not easily scalable.

Lahiri et al., cf. [35], reduce the research to *"assumed-to-be but cannot-be-proved-to-be-equivalent"* pieces of binary code. In general, the attempt is useful for us. Unfortunately, the used assumption of a variable mapping makes the legacy to be not so attractive.

### *2.7. Locating Compiler Bugs*

Another concept that could be useful for our purpose is a technique that handles compiled versions of the same procedure from different compilers, as presented in [36] (More information about the concept and the similar ones may be found in [37].). The authors compare *Intermediate Language* code obtained from different compilers to root-cause compiler bugs. To do so, the approach uses semantic tools. However, it is not applicable to our purpose. The approach is aimed to find full equivalence rather than some kinds of similarity, which we need. In fact, it was the identification of root causes of compiler bugs. Unfortunately, this promising approach cannot be easily adopted to our research as well for the following reasons:

(i) The bug detection requires precise equivalence that is not acceptable for our case, where small changes are allowed;
(ii) One of the consequence is a problem of loop handling due to inability to support procedures' segmentation;
(iii) and more ...

### *2.8. Detecting Software Plagiarism*

Techniques for detecting plagiarism in computer code go back to [38]. MOSS (Measure Of Software Similarity) is aimed to detect plagiarism in programming *classes* and is based on ideas, similar to ours: disintegrate programs into fragments and check these fragments for similarity The main difference is in the disintegration and the next coming similarity identification: in Moss, the code is decomposed in resolution of lines and the similarity is detected via an exact syntactic match. In fact, we start from the main ideas of the Moss and extend them significantly.

The authors of [39] proposed and implemented the source code similarity system for plagiarism detection. In this paper, we extend their approach of similarity measurement by adding more features. Moreover, we also extend their approach of final similarity calculation by introducing our optimization machinery.

In [40], a different attempt of code similarity analysers was introduced. The main novelty is the analysis of code clone for plagiarism detectors. In addition, lots of similarity detection techniques and tools are evaluated. The method is based on use of a specific data set. However, it turns out that the obtained optimal configurations strongly depend upon a particular choice of the data set. Our idea to use weighted optimization machinery for detecting plagiarism is also inspired by [41]. However, we do not address the problems, related to decryption.

### *2.9. Using Machine Learning and Clustering Techniques*

We mentioned above that in [29], the authors propose a method for determining even complex lineage for executables. Nonetheless, at its core their method uses linear $n$-grams combined with normalization steps (in this case also normalizing jumps), is inherently flawed due to reliance on the compiler to make the same layout choices. One of the first successful attempts to use machine learning and clustering machinery for efficient detection of plagiarism in homework programs of students was provided in [42]. To do so, the implementation incorporates different plagiarism detection schemes.

In [43], machine learning approach is also used in order to detect code clones. The practical classifier is implemented and it is called *a Filter for Individual user on code Clone Analysis* (Fica) tool. The implementation is based on a comparison of token type sequences of different codes and detection of their similarity. Another name used in the paper is *"term frequency-inverse document frequency"* vector. Fica learns the user opinions concerning these code clones from the classification result.

The method of [44] uses extension of *hierarchical agglomerative clustering*, which is applied to textual descriptions of codes. The obtained results demonstrate success in detecting similarity, based on different "collocation-based" and "dependency-based" features as well as topic modelling. The most recent reviews in the field may be found in [45–47].

### *2.10. Similarity Measures Developed for Code Synthesis Testing*

The authors of [48] are interested in investigation of transformation correctness tests, using comparison of `x86` loop-free snippets. The code is ran on selected inputs. The results are compared as well as the corresponding transitions of the derived state machine. It means that huge number of the runs is needed that is not relevant to our scenaria. That is why, the proposed definition of the distance metric is problematic when we want to compare two code fragments and to detect their similarity. In general, an appropriate definition of such distance metrics is very complicated, cf. [49].

### *2.11. Siamese*

Siamese is a scalable and incremental code clone detection tool proposed in [50]. The main contribution of the paper consists in using different code representations proposed for clones' detection: inverted index, query reduction, and customised ranking, derived from information reconstructions. The approach determines and reports a specific clone type.

## 3. A Motivating Example and Some Similarity Features

We start our feasibility study with an example of results of two compilations of boost libraries. The first compilation was a debug one. To compile as debug, you need to use a debug flag in Linux shell. This instruction is used as input to the ''`make`'' program in order to compile the given code with debug symbols, functions name, variables name, and many more information, see Figure 6.

The second compilation was an optimized one. As it is well-known, optimized compilation tries to minimize or maximize some attributes of an executable computer program. This compilation requires flag ''`-O2`'' as input to ''`make`'' program through boost script. The optimization process drops all "redundant" information from the executable. The optimized Assembly implementation function may be much smaller and different than in the debug mode, compare Figures 6–8.

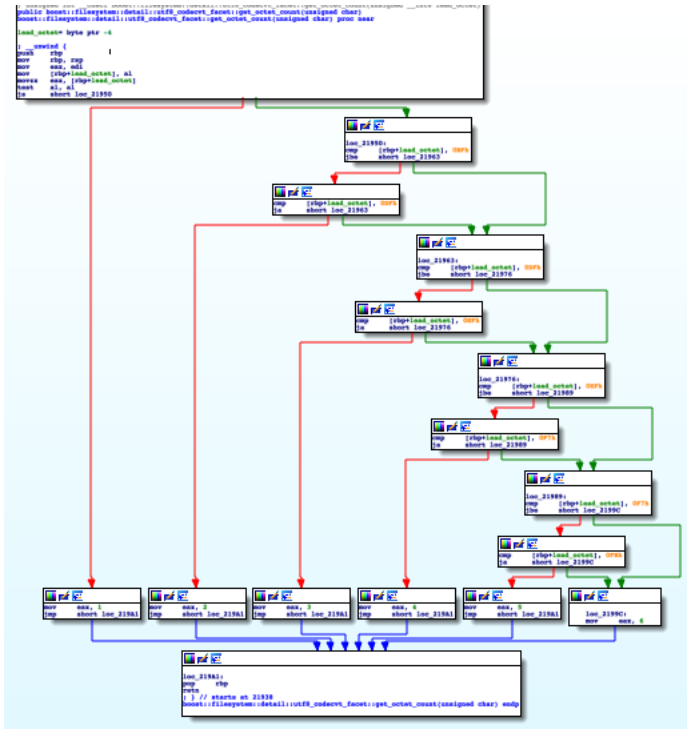


**Figure 6.** Compilation in debug mode.

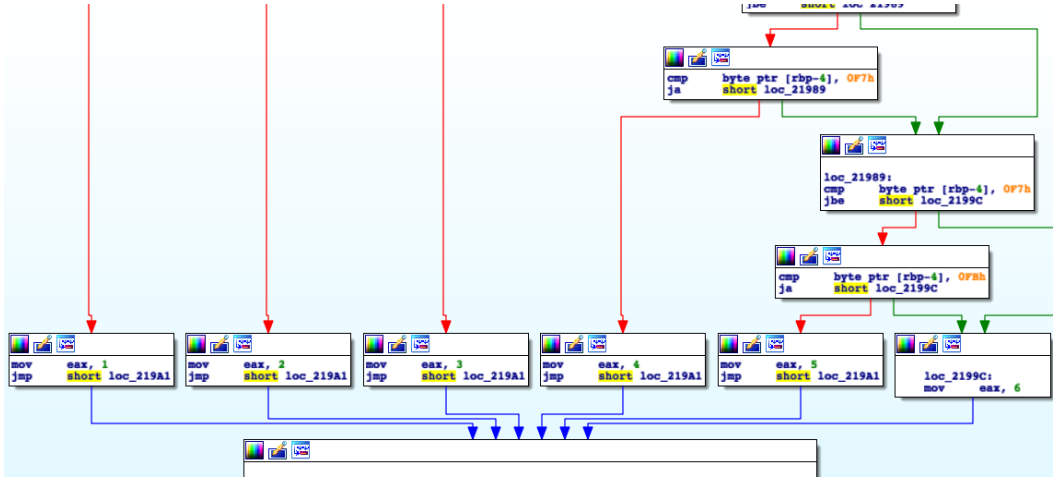



**Figure 7.** Compilation in debug mode (some megascopic view).

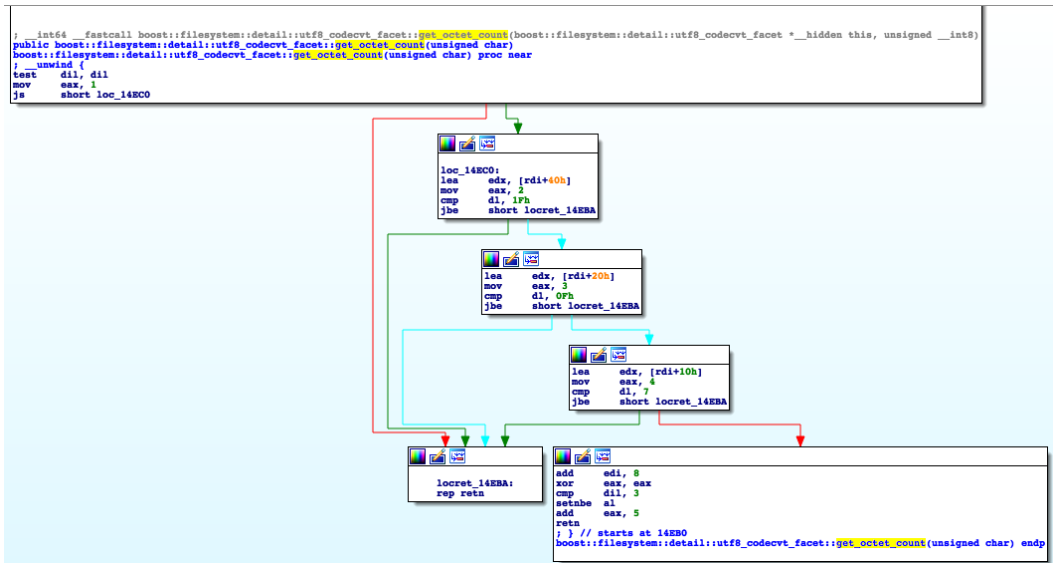



**Figure 8.** Compilation in optimization mode.

Now, we take a real example from boost libraries. We compare between two different results of compilation of the ''`scoped_lock`'' function, compiled with the debug flag, as shown on Figure 9. The optimized result is rather shown on Figure 10 ''`scoped_lock(void const volatile*)`'' function. Now, we propose some features for comparison of two functions: one with debug and the optimized one.

### *3.1. Mnemonic Match*

This feature evaluates the relation of the mnemonic appearance of the instructions in the optimization and the command state in the debug. Committed enumerates the mnemonics, which appear in the optimization, and search for match on their command in the debug. In fact, this feature checks the containment between mnemonics, based on the assumption that the transition between the debug and optimized function deletes or unifies commands in order to save the run time.

In our example, see Figures 9 and 10, the debug function includes the following commands:

- push, **mov, mul, shr, shl, add, sub, lea,** cmp, **jz, jnz, jmp, nop, xchg, test, pause, movzx,** setnz, pop, **retn**.

While the optimized function includes the following commands:

- **mov, mul, shr, shl, add, sub, lea, jz, jnz, jmp, nop, xchg, test, pause, movzx, retn**.

In this case, the feature gives (see the instructions in bold):

$$P_{MnemonicMatch}(x) = \frac{\text{of the containment instructions}}{\text{of instructions} \in \text{the optimized function}} = \frac{16}{16} = 100\%.$$

### *3.2. Command Rare Match*

This feature describes the score, given to the criterion, which evaluates the rare instructions in the optimized mode to the rare instructions in the debug mode. To do so, we must enumerate the mnemonic that appears once and search for match on their instruction between optimized and debug modes.

This feature considers special instructions such as interrupt calls or privilege instruction (ring 0). In the optimized function, the following mnemonic of instructions appears one time, see Figure 11.

```
1.  .text:000000000000074A    push    rbp
2.  .text:000000000000074B    mov     rbp, rsp
3.  .text:000000000000074E    mov     [rbp+this], rdi
4.  .text:0000000000000752    mov     [rbp+addr], rsi
5.  .text:0000000000000756    mov     rcx, [rbp+addr]
6.  .text:000000000000075A    mov     rdx, 0C7CE0C7CE0C7CE0Dh
7.  .text:0000000000000764    mov     rax, rcx
8.  .text:0000000000000767    mul     rdx
9.  .text:000000000000076A    shr     rdx, 5
10. .text:000000000000076E    mov     rax, rdx
11. .text:0000000000000771    shl     rax, 2
12. .text:0000000000000775    add     rax, rdx
13. .text:0000000000000778    shl     rax, 3
14. .text:000000000000077C    add     rax, rdx
15. .text:000000000000077F    sub     rcx, rax
16. .text:0000000000000782    mov     rdx, rcx
17. .text:0000000000000785    shl     rdx, 6
18. .text:0000000000000789    lea     rax, boost__atomics__detail__anonymous_namespace__g_lock_pool
19. .text:0000000000000790    add     rdx, rax
20. .text:0000000000000793    mov     rax, [rbp+this]
21. .text:0000000000000797    mov     [rax], rdx
22. .text:000000000000079A    mov     rax, [rbp+this]
23. .text:000000000000079E    mov     rax, [rax]
24. .text:00000000000007A1    mov     [rbp+var_10], rax
25. .text:00000000000007A5    mov     [rbp+var_20], 2
26. .text:00000000000007AC    mov     eax, [rbp+var_20]
27. .text:00000000000007AF    mov     [rbp+var_1C], eax
28. .text:00000000000007B2    cmp     [rbp+var_1C], 0
29. .text:00000000000007B6    jz      short loc_7DB
30. .text:00000000000007B8    cmp     [rbp+var_1C], 1
31. .text:00000000000007BC    jz      short loc_7DA
32. .text:00000000000007BE    cmp     [rbp+var_1C], 2
33. .text:00000000000007C2    jz      short loc_7D6
34. .text:00000000000007C4    cmp     [rbp+var_1C], 4
35. .text:00000000000007C8    jz      short loc_7D2
36. .text:00000000000007CA    cmp     [rbp+var_1C], 6
37. .text:00000000000007CE    jnz     short loc_7D9
38. .text:00000000000007D0    jmp     short loc_7D3
39. .text:00000000000007D2    nop
40. .text:00000000000007D3    nop
41. .text:00000000000007D4    jmp     short loc_7DB
42. .text:00000000000007D6    nop
43. .text:00000000000007D7    jmp     short loc_7DA
44. .text:00000000000007D9    nop
45. .text:00000000000007DA    nop
46. .text:00000000000007DB    mov     rdx, [rbp+var_10]
47. .text:00000000000007DF    mov     eax, 1
48. .text:00000000000007E4    xchg    al, [rdx]
49. .text:00000000000007E6    test    al, al
50. .text:00000000000007E8    jz      short loc_843
51. .text:00000000000007EA    pause
52. .text:00000000000007EC    mov     rax, [rbp+this]
53. .text:00000000000007F0    mov     rax, [rax]
54. .text:00000000000007F3    mov     [rbp+var_8], rax
55. .text:00000000000007F7    mov     [rbp+var_18], 0
56. .text:00000000000007FE    mov     eax, [rbp+var_18]
57. .text:0000000000000801    mov     [rbp+var_14], eax
58. .text:0000000000000804    cmp     [rbp+var_14], 0
59. .text:0000000000000808    jz      short loc_82D
60. .text:000000000000080A    cmp     [rbp+var_14], 1
61. .text:000000000000080E    jz      short loc_82C
62. .text:0000000000000810    cmp     [rbp+var_14], 2
63. .text:0000000000000814    jz      short loc_828
64. .text:0000000000000816    cmp     [rbp+var_14], 4
65. .text:000000000000081A    jz      short loc_824
66. .text:000000000000081C    cmp     [rbp+var_14], 6
67. .text:0000000000000820    jnz     short loc_82B
68. .text:0000000000000822    jmp     short loc_825
69. .text:0000000000000824    nop
70. .text:0000000000000825    nop
71. .text:0000000000000826    jmp     short loc_82D
72. .text:0000000000000828    nop
73. .text:0000000000000829    jmp     short loc_82C
74. .text:000000000000082B    nop
75. .text:000000000000082C    nop
76. .text:000000000000082D    mov     rax, [rbp+var_8]
77. .text:0000000000000831    movzx   eax, byte ptr [rax]
78. .text:0000000000000834    test    al, al
79. .text:0000000000000836    setnz   al
80. .text:0000000000000839    test    al, al
81. .text:000000000000083B    jz      loc_79A
82. .text:0000000000000841    jmp     short loc_7EA
83. .text:0000000000000843    nop
84. .text:0000000000000844    pop     rbp
85. .text:0000000000000845    retn
```

**Figure 9.** ‘‘`scoped_lock`’’ function compiled with the debug flag.

```
1.  .text:0000000000000750    mov     rax, rsi
2.  .text:0000000000000753    mov     rdx, 0C7CE0C7CE0C7CE0Dh
3.  .text:000000000000075D    mov     ecx, 1
4.  .text:0000000000000762    mul     rdx
5.  .text:0000000000000765    shr     rdx, 5
6.  .text:0000000000000769    lea     rax, [rdx+rdx*4]
7.  .text:000000000000076D    lea     rax, [rdx+rax*8]
8.  .text:0000000000000771    sub     rsi, rax
9.  .text:0000000000000774    lea     rax, boost::atomics::detail::`anonymous
     namespace'::g_lock_pool
10. .text:000000000000077B    shl     rsi, 6
11. .text:000000000000077F    add     rax, rsi
12. .text:0000000000000782    mov     [rdi], rax
13. .text:0000000000000785    nop     dword ptr [rax]
14. .text:0000000000000788    mov     edx, ecx
15. .text:000000000000078A    xchg    dl, [rax]
16. .text:000000000000078C    test    dl, dl
17. .text:000000000000078E    jz      short locret_7A0
18. .text:0000000000000790    pause
19. .text:0000000000000792    mov     rax, [rdi]
20. .text:0000000000000795    movzx   edx, byte ptr [rax]
21. .text:0000000000000798    test    dl, dl
22. .text:000000000000079A    jnz     short loc_790
23. .text:000000000000079C    jmp     short loc_788
24. .text:000000000000079E    align 20h
25. .text:00000000000007A0    rep retn
```

**Figure 10.** ‘‘`scoped_lock`’’ function compiled with optimization.

```
1. mul     rdx
2. shr     rdx, 5
3. sub     rsi, rax
4. shl     rsi, 6
5. add     rax, rsi
6. nop     dword ptr [rax]
7. xchg    dl, [rax]
8. jz      short locret_7A0   // ignore
9. pause
10. movzx   edx, byte ptr [rax]
11. jnz     short loc_790    // ignore
12. jmp     short loc_788    // ignore
13. retn
```

**Figure 11.** The mnemonic of instructions that appears one time in the optimized function.

Here, we note that:

1. We will always drop the jump instructions because as a rule the target jump will be different, and it will always reduce the probability of imagination.
2. We will always treat registers as the same variable this grade will related as 0.5 (instead 1).

At this point, we search for the similar instructions in the debug and calculate the probability:

$$P_{CommandRareMatch}(x) = \frac{\text{of } \times \text{ rare instruction } \in \text{ the optimized exist } \in \text{ the debug}}{\text{of rare mnemonic instructions } \in \text{ the optimized function}} = \frac{8.5}{10} = 85\%.$$

### *3.3. Constant Match*

This feature describes the grade, given for the similarity between fixed values and mnemonic between the optimized and the debug mode. It draws a similarity between the functions of an encryption library and/or any other function, which uses a special constant to calculate something. The reason of attaching of the mnemonic is to reduce noise that can be added unnecessarily. The calculation is performed by collecting all the instructions with constants in the optimized function and then comparing them to the debug function. The instructions that included constants in the optimized function, are shown on Figure 12.

Here you are the precise calculation of the feature for our example:

$$P_{ConstantMatch}(x) = \frac{\text{of } \times \text{ exist } \in \text{ the debug}}{\text{of constants instructions}} = \frac{3}{4} = 75\%.$$

```
1. mov     rdx, 0C7CE0C7CE0C7CE0Dh
2. mov     ecx, 1
3. shr     rdx, 5
4. shl     rsi, 6
```

**Figure 12.** Commands that included constants in the optimized function.

### *3.4. Nested Function Match*

This feature describes the grade, given for the similarity in function calls between the debug function and the optimized function. It is important to note that this feature assumes all nested functions identified therefore, needed to run this in recursive mode. In this case, we have no calls: therefore the probability is 1:

$$P_{NestedFunctionMatch}(x) = \frac{\text{of } \times \text{ exist } \in \text{ the debug } \wedge \text{ the optimized}}{\text{of call nested function } \in \text{ the debug}}.$$

To be clear, the following example describes the situation. Let us take the "`printf`" function (from `uClibc` library). This function includes calls to two internal functions. The debug function of "`printf`" is shown on Figure 13. The optimized function of "`printf`" is rather shown on Figure 14. Both, the debug function and the optimized functions, contain

two calls. Just compare, `vfprintf` and `__GI_vfprintf`. Moreover, the call to `sub_10CE9` is previously detected as `__x86_get_pc_thunk_ax`. It means that

$$P_{NestedFunctionMatch}(x) = \frac{2}{2} = 100\%$$

in this case.

```
1. push    ebp
2. mov     ebp, esp
3. sub     esp, 18h
4. call    __x86_get_pc_thunk_ax
5. add     eax, 744ABh
6. lea     edx, [ebp+arg_4]
7. mov     [ebp+arg], edx
8. mov     edx, [ebp+arg]
9. mov     eax, ds:(stdout_ptr 0B9C7Ch)[eax]
10. mov    eax, dword ptr ds:(_GLOBAL_OFFSET_TABLE_
11. sub    esp, 4
12. push   edx             ; arg
13. push   [ebp+format]    ; format
14. push   eax             ; stream
15. call   __GI_vfprintf
16. add    esp, 10h
17. mov    [ebp+rv], eax
18. mov    eax, [ebp+rv]
19. leave
20. retn
```

**Figure 13.** The debug function of "`printf`".

```
1. call    sub_10CE9
2. add     eax, 57F37h
3. sub     esp, 0Ch
4. mov     eax, ds:(stdout_ptr 88C7Ch)[eax]
5. lea     edx, [esp+0Ch+arg_4]
6. sub     esp, 4
7. push    edx             ; int
8. push    [esp+14h+arg_0] ; void *
9. push    dword ptr ds:(tbyte_88C7C 88C7Ch)[eax] ;
10. call   vfprintf
11. add    esp, 1Ch
12. retn
```

**Figure 14.** The optimized function of "`printf`".

### *3.5. Jump Match*

This feature describes the grade, given for similarity in all conditional jumps between the debug function and the optimized function. This assume is based on avoiding `jump` prediction by using the carry in the flag register.

The following example finds the minimum of two unsigned numbers: `if ( b < a ) a = b;`.

```
1.  sub  ebx,eax
2.  sbb  ecx,ecx
3.  and  ecx,ebx
4.  add  eax,ecx
```

The next example chooses between two numbers: `if ( a!=0) a = b; else a = c;`.

```
1.  cmp  eax,1
2.  sbb  eax,eax
3.  xor  ecx,ebx
4.  add  eax,ecx
5.  xor  eax,ebx
```

Let us take an example of "`pathconf`" function: Its debug function is shown on Figure 15 and its optimized function is shown on Figure 16. We define $P_{JumpMatch}(x)$ as:

$$P_{JumpMatch}(x) = \frac{\text{of conditional jumps } \in \text{ the optimized}}{\text{of conditional jumps } \in \text{ the debug}} = \frac{6}{7} = 0.85\%.$$

Table 1 summarizes the jump instructions in these functions.

**Table 1.** Summary of different jumps in debug and optimized functions.

| | ja | jnz | js | jns | jz | Total |
|---|---|---|---|---|---|---|
| Optimized | 1 | 2 | 2 | 1 | 0 | 6 |
| Debug | 1 | 2 | 1 | 1 | 2 | 7 |

```
push    ebp
mov     ebp, esp
push    ebx
sub     esp, 64h
call    __x86_get_pc_thunk_bx
add     ebx, 4D05Dh
mov     eax, [ebp+path]
movzx   eax, byte ptr [eax]
test    al, al
jnz     short loc_6CC46
mov     eax, ds:(off_B9F34 - 0B9C7Ch)[ebx]
mov     dword ptr gs:[eax], 2
mov     eax, 0FFFFFFFFh
jmp     loc_6CD70
cmp     [ebp+name], 13h
ja      short loc_6CC5D
mov     eax, [ebp+name]
shl     eax, 2
mov     eax, ds:(off_9FF94 - 0B9C7Ch)[eax+ebx]
add     eax, ebx
jmp     eax
mov     eax, ds:(off_B9F34 - 0B9C7Ch)[ebx]
mov     dword ptr gs:[eax], 16h
mov     eax, 0FFFFFFFFh
jmp     loc_6CD70
mov     eax, 7Fh
jmp     loc_6CD70
mov     eax, 0FFh
jmp     loc_6CD70
mov     eax, 0FFh
jmp     loc_6CD70
mov     eax, ds:(off_B9F34 - 0B9C7Ch)[ebx]
mov     eax, gs:[eax]
mov     [ebp+save_errno], eax
sub     esp, 8
lea     eax, [ebp+st]
push    eax
push    [ebp+path]
call    statfs_0
add     esp, 10h
test    eax, eax
jns     short loc_6CCE2
mov     eax, ds:(off_B9F34 - 0B9C7Ch)[ebx]
mov     eax, gs:[eax]
cmp     eax, 26h ; '&'
jnz     short loc_6CCD8
mov     eax, ds:(off_B9F34 - 0B9C7Ch)[ebx]
mov     edx, [ebp+save_errno]
mov     gs:[eax], edx
mov     eax, 0FFh
jmp     loc_6CD70
mov     eax, 0FFFFFFFFh
jmp     loc_6CD70
mov     eax, dword ptr [ebp+st.st_rdev+4]
jmp     loc_6CD70
mov     eax, 1000h
jmp     short loc_6CD70
mov     eax, 1000h
jmp     short loc_6CD70
mov     eax, 0
jmp     short loc_6CD70
mov     eax, 1
jmp     short loc_6CD70
mov     eax, 0
jmp     short loc_6CD70
mov     eax, 0FFFFFFFFh
jmp     short loc_6CD70
sub     esp, 8
lea     eax, [ebp+st]
push    eax
push    [ebp+path]
call    stat_0
add     esp, 10h
test    eax, eax
js      short loc_6CD48
mov     eax, [ebp+st.st_mode]
and     eax, 0F000h
cmp     eax, 8000h
jz      short loc_6CD4F
mov     eax, [ebp+st.st_mode]
and     eax, 0F000h
cmp     eax, 6000h
jz      short loc_6CD4F
mov     eax, 0FFFFFFFFh
jmp     short loc_6CD70
mov     eax, 1
jmp     short loc_6CD70
mov     eax, 0FFFFFFFFh
jmp     short loc_6CD70
mov     eax, 0FFFFFFFFh
jmp     short loc_6CD70
mov     eax, 20h
jmp     short loc_6CD70
mov     eax, 0FFFFFFFFh
mov     ebx, [ebp+var_4]
leave
retn
```

**Figure 15.** The debug function of "`pathconf`".

```
1. .text:00046FA3                push    edi
2. .text:00046FA4                push    esi
3. .text:00046FA5                push    ebx
4. .text:00046FA6                call    sub_F96A
5. .text:00046FAB                add     ebx, 41CD1h
6. .text:00046FB1                sub     esp, 60h
7. .text:00046FB4                mov     ecx, [esp+6Ch+arg_0]
8. .text:00046FB8                mov     edx, [esp+6Ch+arg_4]
9. .text:00046FBC                test    ecx, ecx
10..text:00046FBE                js      loc_4709F
11..text:00046FC4                test    edx, edx
12..text:00046FC6                mov     eax, 7Fh
13..text:00046FCB                jnz     short loc_46FD4
14..text:00046FCD                add     esp, 60h
15..text:00046FD0                pop     ebx
16..text:00046FD1                pop     esi
17..text:00046FD2                pop     edi
18..text:00046FD3                retn
19..text:00046FD4                cmp     edx, 13h
20..text:00046FD7                ja      loc_47088
21..text:00046FDD                mov     eax, ds:(off_65174 -
    88C7Ch)[ebx+edx*4]
22..text:00046FE4                add     eax, ebx
23..text:00046FE6                jmp     eax
24..text:00046FE8                mov     eax, 0FFFFFFFFh
25..text:00046FED                jmp     short loc_46FCD
26..text:00046FEF                xor     eax, eax
27..text:00046FF1                jmp     short loc_46FCD
28..text:00046FF3                mov     eax, 1000h
29..text:00046FF8                jmp     short loc_46FCD
30..text:00046FFA                mov     eax, 0FFh
31..text:00046FFF                jmp     short loc_46FCD
32..text:00047001                mov     eax, 20h
33..text:00047006                jmp     short loc_46FCD
34..text:00047008                sub     esp, 8
35..text:0004700B                lea     eax, [esp+74h+var_64]
36..text:0004700F                push    eax
37..text:00047010                push    ecx
38..text:00047011                call    fstat
39..text:00047016                add     esp, 10h
40..text:00047019                test    eax, eax
41..text:0004701B                js      short loc_46FE8
42..text:0004701D                mov     eax, [esp+6Ch+var_54]
43..text:00047021                and     eax, 0F000h
44..text:00047026                sub     eax, 6000h
45..text:0004702B                test    eax, 0FFFFD000h
46..text:00047030                setnz   al
47..text:00047033                movzx   eax, al
48..text:00047036                neg     eax
49..text:00047038                or      eax, 1
50..text:0004703B                jmp     short loc_46FCD
51..text:0004703D                mov     eax, 1
52..text:00047042                jmp     short loc_46FCD
53..text:00047044                sub     esp, 8
54..text:00047047                mov     esi, ds:(off_88F30 -
    88C7Ch)[ebx]
55..text:0004704D                lea     eax, [esp+74h+var_64]
56..text:00047051                mov     edi, gs:[esi]
57..text:00047054                push    eax
58..text:00047055                push    ecx
59..text:00047056                call    fstatfs
60..text:0004705B                mov     edx, eax
61..text:0004705D                add     esp, 10h
62..text:00047060                mov     eax, [esp+6Ch+var_40]
63..text:00047064                test    edx, edx
64..text:00047066                jns     loc_46FCD
65..text:0004706C                cmp     dword ptr gs:[esi], 26h
66..text:00047070                mov     eax, 0FFFFFFFFh
67..text:00047075                jnz     loc_46FCD
68..text:0004707B                mov     gs:[esi], edi
69..text:0004707E                mov     eax, 0FFh
70..text:00047083                jmp     loc_46FCD
71..text:00047088                mov     eax, ds:(off_88F30 -
    88C7Ch)[ebx]
72..text:0004708E                mov     dword ptr gs:[eax], 16h
73..text:00047095                mov     eax, 0FFFFFFFFh
74..text:0004709A                jmp     loc_46FCD
75..text:0004709F                mov     eax, ds:(off_88F30 -
    88C7Ch)[ebx]
76..text:000470A5                mov     dword ptr gs:[eax], 9
77..text:000470AC                mov     eax, 0FFFFFFFFh
```

**Figure 16.** The optimized function of "`pathconf`".

## 4. Analyzing the Correctness of the Similarity Features

In Section 3, we introduced five similarity features: mnemonic match, instruction rare match, constant match, nested function match, and illustrated them on a real-life example. Now, we note that in general everyone may add more and more such features. The ability poses several questions for each of such $P_{MyLovely_\ell}$ features:

1. Does $P_{MyLovely_\ell}$ feature really reflect any similarity between codes?
2. Given several (already five defined) features, which give different results for the same pieces of codes: how do we define the impact of each one on our final (binary:*yes* or *no*) decision if these pieces of codes in fact are similar?
3. Given two $P_{MyLovely_\ell}$ and $P_{MyLovely_\wp}$ such features: do they in fact reflect the same similarity feature or rather different ones?

4. Given a set of $P_{MyLovely_\ell}$: does it contain contradictions, by means that one feature witnesses similarity while another one does not (On the other hand, having different features giving different results because focusing on different aspects of similarity, seems to be sometimes even desirable in order to obtain the best possible precision in detecting similarities.)?
5. Given a set of $P_{MyLovely_\ell}$: does it cover all scenaria of similarities in codes? an so on ...

In this section, we try to define an approach to answer the first question: *Does* $P_{MyLovely_\ell}$ *feature really reflect any similarity between codes?* To do so, we propose a statistical approach. We take functions from `uClibc` library and compile them in two modes, which produces pairs of identical functions (we call them *touchstones*) and evaluate the introduced features on each pair. If a feature is "good" it should report similarity in each such a pair as, in fact, this is the same function. The obtained results are shown on Figures 17–21. On the figures, axe $x$ shows intervals of the corresponding features, and axe $y$ shows the numbers of the pairs, which fall in the corresponding interval.

Analyzing the figures, for example Figure 21, it may be easily seen that the *jump match feature* is a feature that can be really matched in most cases. So, it can be said that it is a quality feature and yet the *nested function match feature* (Figure 20) does not reflect reality and it is not conclusive that it can be taken as part of the similarity identification. The question that arises immediately is: do the results, shown on Figures 17–21, depend upon the chosen set of *touchstones*? If so, what should be the *golden set* of the *touchstones* if any?

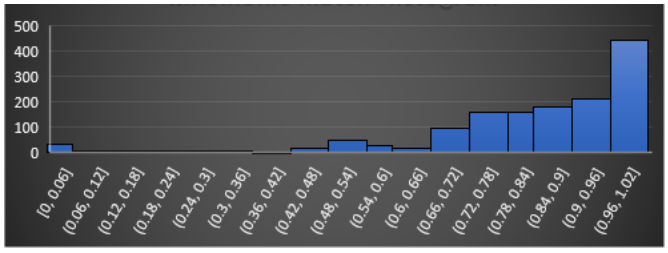



**Figure 17.** The mnemonic match histogram.

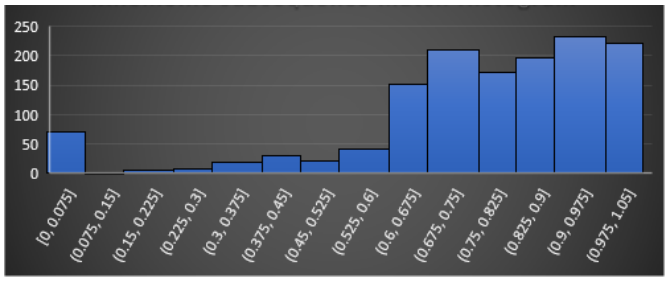



**Figure 18.** The instruction rare match histogram.

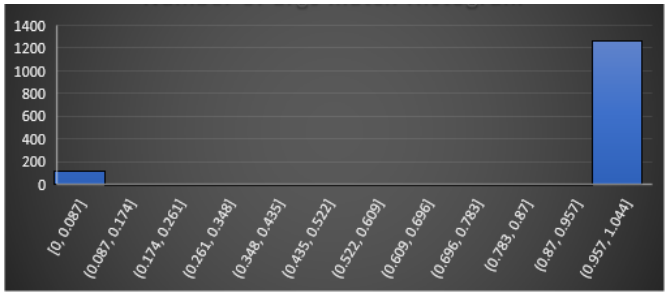



**Figure 19.** The constant match histogram.

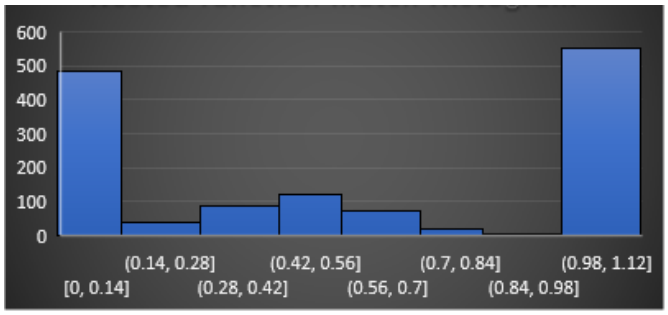



**Figure 20.** The nested function match histogram.

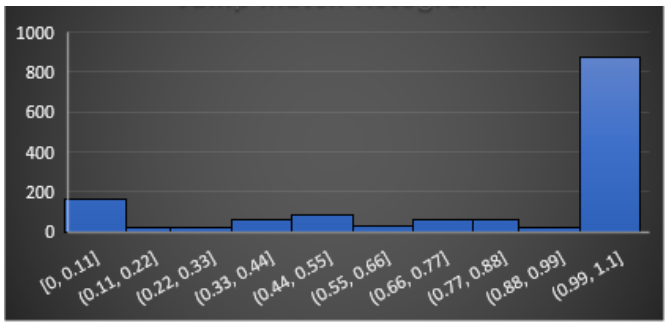



**Figure 21.** The jump match histogram.

## 5. Formulating the Similarity Detection Problem as an Optimization Problem

At this step, we have, see Figure 5:

- the source code, composed of $N$ functions: $function_1, \ldots, function_N$;
- results of compilation the functions in debug mode $function_{D_1}, \ldots, function_{D_N}$;
- results of compilation the functions in optimization mode $function_{O_1}, \ldots, function_{O_N}$;
- $K$ features like those, introduced in Section 3, which are defined to be meaningful for the similarity definition according to the statistic approach of Section 4;
- results $\overrightarrow{p}_k$ of evaluation of each such a feature $p_k$ on each pair of functions $function_{D_n}$ and $function_{O_n}$.

We recall that our aim is to answer the final (binary: *yes* or *no*) question if these pieces of codes in fact are similar using the values of $\overrightarrow{p}_k$. To do so, we should define the impact factor of each feature on the final decision. We formulate the last problem as a mathematical optimization problem. To do so, we create a formula, which reflects the impact of every chosen feature on the final decision, based on weights of the features.

Till now, in the general case, we defined a set of $N$ $function_i$. At this moment, we may take the debug and the optimized codes of "`uClibc`" and calculate the values of the already defined features between every pair's functions (optimize and debug). We defined $\overrightarrow{p}_k$ $(k = 1, \ldots, K)$ to compare the values of each feature $k$ on each pair of the compiled functions in order to define the similarity of the codes. Now, we want to use the values of $\overrightarrow{p}_k$ in order to define an optimization problem. Here, we choose to deal with a weighted optimization problem.

### 5.1. Definition of Impact Factors of Features

Given results $\overrightarrow{p}_k$ of evaluation of each feature $p_k$ on each pair of functions $function_{D_n}$ and $function_{O_n}$. We want to define weights: $w_k$ in order to solve the similarity detection problem. How do we define the $K$ weights?

First of all, we note that every $function_{D_n}$, $function_{O_n}$ as well as $\overrightarrow{p}_k$ from Figure 5 can be combined via the matrix, shown on Figure 22. It leads us to the following general formulation $similarity(function_{D_i}, function_{O_j})$ :

$$x_{ij} = \overrightarrow{w}^t \cdot \overrightarrow{p}(function_{D_i}, function_{O_j}) = \sum_{k=1}^{K} w_k \cdot p_k^{i,j},$$

where $p_k^{i,j}$ is the value of feature $p_k$ on a pair of functions: $function_{D_i}$ and $function_{O_j}$.

We define $\overrightarrow{p}^{i,j}$ to be the vector of the values of $K$ features on the pair of functions $function_{D_i}$ and $function_{O_j}$. Here, $\overrightarrow{w}$ is the vector of length $K$ of the weights of the $\overrightarrow{p}^{i,j}$: every feature has its own weight partner. There are $K$ such pairs with $K$ undefined values of $\overrightarrow{w}$. The last definition provides us with a 2D-matrix ($1 \leq i \leq N$ and $1 \leq j \leq N$)

$$x_{ij} = similarity(function_{D_i}, function_{O_j}) :$$

one value of $\sum_{k=1}^{K} w_k \cdot p_k^{i,j}$ for each pair $function_{D_i}$, $function_{O_j}$, where all the values are positive.

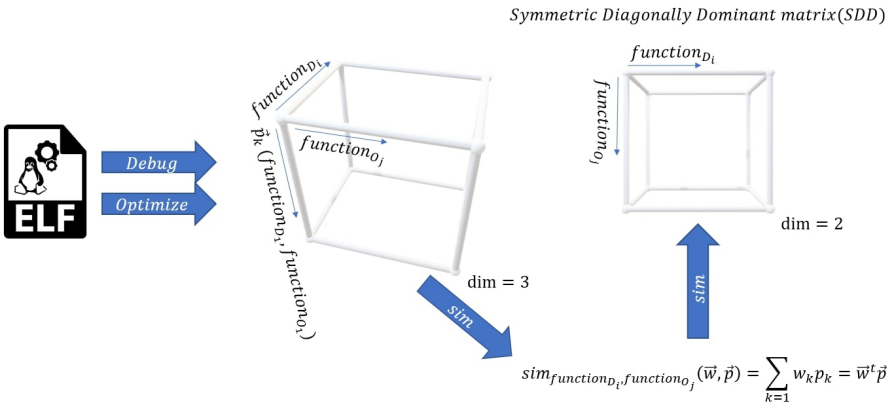



**Figure 22.** Extraction of the Symmetric Diagonally Dominant (SDD) matrix $x_{ij}$.

Now, we formulate our optimization problem in terms of the defined 2D-matrix of $x_{ij}$. In fact: What do we expect from the 2D-matrix of $x_{ij}$? We definitely want it do be: symmetric $x_{ij} = x_{ji}$ and diagonally dominant $x_{i,i} \geq \sum_{j \neq i} x_{ij}$ or in other words $\delta_i = x_{i,i} - \sum_{j \neq i} x_{ij} \geq 0$ (We note that if there are two functions $i$ and $j$ with the same codes it means that our library has redundant functions and the last condition is not satisfiable.).

In fact, $x_{ij} = \overrightarrow{w}^t \overrightarrow{p}^{i,j}$. The diagonal dominant requirement $\delta_i$ gives for each $1 \leq i \leq N$:

$$\delta_i \geq 0$$

or

$$\overrightarrow{w}^t \overrightarrow{p}^{i,i} - \sum_{j \neq i} \overrightarrow{w}^t \overrightarrow{p}^{i,j} \geq 0$$

or

$$\overrightarrow{w}^t (\overrightarrow{p}^{i,i} - \sum_{j \neq i} \overrightarrow{p}^{i,j}) \geq 0.$$

We expand the last condition even more. We define $(1 \leq n \leq N)$:

$$\delta_n = \overrightarrow{w}^t (\overrightarrow{p}^{n,n} - \sum_{j \neq n} \overrightarrow{p}^{n,j})$$

and we sum them: $\sum_{n=1}^{N} \delta_n$. Our expanded requirement is as follows (The intuition is that we want some kind of "comparability" of the $\delta_i$'s: almost identical contribution of each feature to the final decision about the codes similarity.)

$$\delta_i - \sum_{n=1, n \neq i}^{N} \delta_n \geq C_i.$$

By substitution, we receive:

$$\overrightarrow{w}^t\left(\overrightarrow{p}^{i,i}-\sum_{j\neq i}\overrightarrow{p}^{i,j}\right)-\sum_{n=1,n\neq i}^{N}\overrightarrow{w}^t\left(\overrightarrow{p}^{n,n}-\sum_{n\neq j}\overrightarrow{p}^{n,j}\right)\geq C_i$$

$$\overrightarrow{w}^t\left(\left(\overrightarrow{p}^{i,i}-\sum_{j\neq i}\overrightarrow{p}^{i,j}\right)-\sum_{n=1,n\neq i}^{N}\left(\overrightarrow{p}^{n,n}-\sum_{n\neq j}\overrightarrow{p}^{n,j}\right)\right)\geq C_i$$

$$\overrightarrow{w}^t\left(\overrightarrow{p}^{i,i}-\overrightarrow{p}^{i,i}-\left(\sum_{j\neq i}\overrightarrow{p}^{i,j}-\overrightarrow{p}^{i,i}\right)-\sum_{n=1,n\neq i}^{N}\left(\overrightarrow{p}^{n,n}-\sum_{n\neq j}\overrightarrow{p}^{n,j}\right)\right)\geq C_i$$

$$\overrightarrow{w}^t\left(\overrightarrow{p}^{i,i}-\left(\sum_{j\neq i}\overrightarrow{p}^{i,j}-\overrightarrow{p}^{i,i}\right)-\overrightarrow{p}^{i,i}-\sum_{n=1,n\neq i}^{N}\left(\overrightarrow{p}^{n,n}-\sum_{n\neq j}\overrightarrow{p}^{n,j}\right)\right)\geq C_i$$

$$\overrightarrow{w}^t\left(\overrightarrow{p}^{i,i}-\left(\sum_{j\neq i}\overrightarrow{p}^{i,j}-\overrightarrow{p}^{i,i}\right)-\sum_{n=1,n\neq i}^{N}\left(\overrightarrow{p}^{n,n}-\left(\sum_{n\neq j}\overrightarrow{p}^{n,j}-\overrightarrow{p}^{n,n}\right)\right)\right)\geq C_i$$

$$\overrightarrow{w}^t\left(2\cdot\overrightarrow{p}^{i,i}-\sum_{j}\overrightarrow{p}^{i,j}-\sum_{n\neq i}\left(2\cdot\overrightarrow{p}^{n,n}-\sum_{j}\overrightarrow{p}^{i,j}\right)\right)\geq C_i$$

$$\overrightarrow{w}^t\left(2\cdot\overrightarrow{p}^{i,i}-\sum_{j}\overrightarrow{p}^{i,j}-\left(\sum_{n}\left(2\cdot\overrightarrow{p}^{n,n}-\sum_{j}\overrightarrow{p}^{i,j}\right)-\left(2\cdot\overrightarrow{p}^{i,i}-\sum_{j}\overrightarrow{p}^{i,j}\right)\right)\right)\geq C_i$$

$$\overrightarrow{w}^t\left(2\cdot\overrightarrow{p}^{i,i}-\sum_{j}\overrightarrow{p}^{i,j}-\sum_{n}\left(2\cdot\overrightarrow{p}^{n,n}-\sum_{j}\overrightarrow{p}^{i,j}\right)+2\cdot\overrightarrow{p}^{i,i}-\sum_{j}\overrightarrow{p}^{i,j}\right)\geq C_i$$

$$\overrightarrow{w}^t\left(2\cdot\overrightarrow{p}^{i,i}-\sum_{j}\overrightarrow{p}^{i,j}-\sum_{n}2\cdot\overrightarrow{p}^{n,n}+\sum_{n}\sum_{j}\overrightarrow{p}^{i,j}+2\cdot\overrightarrow{p}^{i,i}-\sum_{j}\overrightarrow{p}^{i,j}\right)\geq C_i$$

$$\overrightarrow{w}^t\left(2\cdot\overrightarrow{p}^{i,i}-\sum_{j}\overrightarrow{p}^{i,j}-\sum_{n}2\cdot\overrightarrow{p}^{n,n}+K\cdot\sum_{j}\overrightarrow{p}^{i,j}+2\cdot\overrightarrow{p}^{i,i}-\sum_{j}\overrightarrow{p}^{i,j}\right)\geq C_i$$

$$\overrightarrow{w}^t\left(4\cdot\overrightarrow{p}^{i,i}-4\cdot\sum_{j}\overrightarrow{p}^{i,j}+K\cdot\sum_{j}\overrightarrow{p}^{i,j}\right)\geq C_i$$

$$\overrightarrow{w}^t\left(4\cdot\overrightarrow{p}^{i,i}+(K-4)\cdot\sum_{j}\overrightarrow{p}^{i,j}\right)\geq C_i.$$

We naturally denote:

$$\kappa_i=4\cdot\overrightarrow{p}^{i,i}+(K-4)\cdot\sum_{j}\overrightarrow{p}^{i,j},$$

where each $\kappa_i$ ($1\leq i\leq N$) is a vector of length $K$. Now, for each $1\leq i\leq N$, we have:

$$\overrightarrow{w}^t\cdot\kappa_i\geq C_i.$$

We define vector $\overrightarrow{\mathrm{Y}}$ of $\kappa_1,\kappa_2,\ldots,\kappa_N$, such that each $\kappa_i$ is of length $K$ and require: $\overrightarrow{w}^t\cdot\overrightarrow{\mathrm{Y}}\geq\mathtt{C}$, where $\mathtt{C}$ is a multi-valued and composed of $C_i$'s, $1\leq i\leq N$. If we want to avoid the multi-objective approach then we may reduce the problem to the case, where all values $C_i$'s, $1\leq i\leq N$ are equal.

Recall that we are looking for weights: it means that $w_k\geq 0$ and moreover $\sum_{k=1}^{K}w_k=1$. Now, our optimization problem may be formulated in the following way:

*Given:*

- *the source code, composed of N functions:* $function_1, \ldots, function_N$;
- *results of compilation the functions in debug mode* $function_{D_1}, \ldots, function_{D_N}$;
- *results of compilation the functions in optimization mode* $function_{O_1}, \ldots, function_{O_N}$;
- *K pre-defined features;*
- *results* $p_k^{i,j}$ *of evaluation of feature k on each pair of functions* $function_{D_i}$ *and* $function_{O_j}$;
- *N vectors* $\kappa_i$ *each of length K, defined as* $\kappa_i = 4 \cdot \overrightarrow{p}^{i,i} + (K-4) \cdot \sum_j \overrightarrow{p}^{i,j}$;
- $\overrightarrow{\mathrm{Y}}$ *a vector of length N of* $\kappa_1, \kappa_2, \ldots, \kappa_N$;
- $\complement$ *a vector of length N, composed of* $C_i$*'s,* $1 \leq i \leq N$*, which may be required to be equal.*

*We want to minimize the scalar values of weights* $w_k$ $(1 \leq k \leq K)$*, which maximize* $\complement$*, such that:*

1. $w_k \geq 0$;
2. $\sum_{k=1}^{K} w_k = 1$;
3. $\overrightarrow{w}^t \cdot \overrightarrow{\mathrm{Y}} - \complement \geq 0$.

### *5.2. Examining the Definition of the Impact Factors of Features*

In general, it is strongly desirable to quantify the quality of the used features and therefore, statistics were performed accordingly. To do so, we examined each of the presented features on identical functions, predefined ahead, as shown on Figures 17– 21.

The implementation of the proposed optimization approach is done in `MATLAB`. We implemented the proposed optimization approach using `fminimax` optimization function of `MATLAB` with the corresponding use of `options`, initialization and limitations. First of all, we calculated $\overrightarrow{\mathrm{Y}}$ of $\kappa_1, \kappa_2, \ldots, \kappa_N$, with $\kappa_i$ of length $K$. Then, we coded the optimization problem, as defined above. Testing of the results is done by calculating the grade of similarity for each pair of functions $i$ and $j$ and checking if the results are equal to real data.

In order to examine our approach on a real-life example, we took `uClibc`, which is a relatively small `C` standard library intended for Linux kernel-based operating systems for embedded systems and mobile devices. Our approach demonstrated 93% of success: 851/915.

## 6. Proof of Concept

The proof of concept implementation is done in Python language via IDA, that is a Interactive DisAssembler (IDA) for computer software, which generates `MATLAB` and Assembly language source code from machine-executable code. IDAPython is an IDA plugin. It allows writing scripts for IDA in the Python programming language. IDAPython gives complete access to the IDA API as well as any installed Python module.

Our implementation is based on the proposed approach, which uses features, approved to be meaningful for definition of the similarity, as well as their weights, which determine their impact factor on the finite decision: similar or not the codes under consideration. We use the following notation:

- $func_{D_i}$ denotes the debug function number $i$ as desirable to be identified;
- $func_{O_j}$ denotes the optimized function number $j$ as desirable to be checked;
- $p_{k(func_{D_i}, func_{O_j})}$ denotes the probability of similarity via feature number $k$, given two functions $func_{D_i}$ and $func_{O_j}$;
- set *FPrimitives* contains the *primitive* functions, i.e., functions, which do not include branches to other internal functions;
- set *FNPrimitives* contains the *non-primitive* functions, i.e., functions, which do include branches to other internal functions;
- $W_k$ denotes the weight, given to feature, number $k$;
- $sim(func_{D_i}, func_{O_j})$ denotes the probability of similarity via all the features, given two functions $func_{D_i}$ and $func_{O_j}$, i.e., $\sum_{k=1}^{K} w_k \cdot p_{k(func_{D_i}, func_{O_j})}$.

The iterative version of the algorithm is shown on Figure 23, while the recursive version is rather provided on Figure 24. Full code in MATLAB and Python may by received from the corresponding author via private communication.

```
Given:
                    W⃗ = (w1, w2, ..., wn )
For each func_D(i)
     if func_D(i) is primitive then
          add it to FPremitives
     else
          add it to FNPremitives

For each func_D(i) in FPremitives
     For each func_O(j)
          calculate s = sim(func_D(i), func_O(j))
          append S+=[s]

     Max S
     mark func_D(i) as similar to func_O(j)
     remove func_D(i) and func_O(j) from FPremitives

For each func_D(i) in FNPremitives
     For each func_O(j)
          For each feature k
               calculate s = sim(func_D(i), func_O(j))
               append S+=[s]
          Max S
     mark func_D(i) as similar to func_O(j)
     remove func_D(i) and func_O(j) from FNPremitives
```

**Figure 23.** The iterative algorithm.

```
Given:
                W⃗ = (w1, w2, ..., wn )
Build function call graph
Let's run DFS algorithm on debug Library

for each func_D(i) in DFS
     for each func_O(j)
          calculate s = sim(func_D(i), func_O(j))
          append S+=[s]
     Max S
     mark func_D(i) as like func_O(j)
     mark func_D(i) in three
     remove func_O(j) from debug
```

**Figure 24.** The recursive algorithm.

## 7. Conclusions and Outlook

A binary code of an industrial program might contain procedures, for which a license is required. In order to discover such kinds of software plagiatrism and copyright infringements in the software industry, in this paper, we introduced an extensible approach for examining similarities between binary functions. Once a target architecture and the optimization mode are chosen, the approach can detect similarity with high percent of precision even when the code is compiled using different compilers for any such a choice. However, still both false positives and false negatives might occur even for functions from the same library. It means that the proposed set of similarity features should be extended.

The main idea is to score each function using a set of predetermined features and then find a suitable weight vector that could give impact factor to each feature. After finding the

weight vector, these functions can be identified in binary software packages. This approach eliminates the research process of understanding the functions. To determine similarities in the procedure, an engineering statistical model that guarantees high identification success is used.

**Author Contributions:** Conceptualization, E.V.R. and Z.V.; Methodology, E.V.R. and Z.V.; Software, M.A.; Validation, M.A. and Z.V.; Investigation, M.A. and Z.V.; Resources, M.A.; Writing—review & editing, E.V.R.; Visualization, M.A.; Supervision, Z.V.; Project administration, E.V.R. and Z.V. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Data Availability Statement:** The data presented in this study are available on request from the corresponding author.

**Conflicts of Interest:** Matan Avitan is an employee of Rafael, Advanced Defense Systems Ltd. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest. Rafael, Advanced Defense Systems Ltd.had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

## References

1. Cheung, W.; Ryu, S.; Kim, S. Development nature matters: An empirical study of code clones in JavaScript applications. *Empir. Softw. Eng.* **2015**, *21*, 517–564. [CrossRef]
2. Xu, B.; An, L.; Thung, F.; Khomh, F.; Lo, D. Why reinventing the wheels? An empirical study on library reuse and re-implementation. *Empir. Softw. Eng.* **2020**, *25*, 755–789. [CrossRef]
3. Hu, X.; Li, G.; Xia, X.; Lo, D.; Jin, Z. Deep code comment generation with hybrid lexical and syntactical information. *Empir. Softw. Eng.* **2020**, *25*, 2179–2217. [CrossRef]
4. Thalheim, B. Models and Modelling in Computer Science. In *Logic, Computation and Rigorous Methods-Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday;* Lecture Notes in Computer Science; Raschke, A., Riccobene, E., Schewe, K., Eds.; Springer: Berlin/Heidelberg, Germany, 2021; Volume 12750, pp. 301–325.
5. Thalheim, B. Models: The fourth dimension of computer science. *Softw. Syst. Model.* **2022**, *21*, 9–18. [CrossRef]
6. Bellon, S.; Koschke, R.; Antoniol, G.; Krinke, J.; Merlo, E. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Softw. Eng.* **2007**, *33*, 577–591. [CrossRef]
7. Horwitz, S.; Reps, T.; Binkley, D. Interprocedural Slicing Using Dependence Graphs. *SIGPLAN Not.* **1988**, *23*, 35–46. [CrossRef]
8. Hu, Y.; Fang, Y.; Sun, Y.; Jia, Y.; Wu, Y.; Zou, D.; Jin, H. Code2Img: Tree-Based Image Transformation for Scalable Code Clone Detection. *IEEE Trans. Softw. Eng.* **2023**, *49*, 4429–4442. [CrossRef]
9. Pintér, A.; Szénási, S. Longest Common Subsequence-based Source Code Similarity. In Proceedings of the 2023 IEEE 17th International Symposium on Applied Computational Intelligence and Informatics (SACI), Timisoara, Romania, 23–26 May 2023; pp. 000123–000128.
10. Wang, Y.; Ye, Y.; Wu, Y.; Zhang, W.; Xue, Y.; Liu, Y. Comparison and Evaluation of Clone Detection Techniques with Different Code Representations. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023; pp. 332–344.
11. Zhou, Y.; Chen, J.; Shi, Y.; Chen, B.; Jiang, Z.M.J. An Empirical Comparison on the Results of Different Clone Detection Setups for C-based Projects. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Melbourne, Australia, 14–20 May 2023; pp. 74–86.
12. Alhazami, E.A.; Sheneamer, A.M. Graph-of-Code: Semantic Clone Detection Using Graph Fingerprints. *IEEE Trans. Softw. Eng.* **2023**, *49*, 3972–3988. [CrossRef]
13. Thanoshan, M.; Banujan, K.; Kumara, B.; Prasanth, S.; Li, Z.; Paik, I. Code Clone Detection Using Boosting Algorithms. In Proceedings of the 2023 3rd International Conference on Advanced Research in Computing (ICARC), Belihuloya, Sri Lanka, 23–24 February 2023; pp. 244–249.
14. Zhang, W.; Guo, S.; Zhang, H.; Sui, Y.; Xue, Y.; Xu, Y. Challenging Machine Learning-Based Clone Detectors via Semantic-Preserving Code Transformations. *IEEE Trans. Softw. Eng.* **2023**, *49*, 3052–3070. [CrossRef]
15. Balakrishnan, G.; Reps, T. DIVINE: DIscovering Variables IN Executables. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*; Cook, B., Podelski, A., Eds.; Springer : Berlin/Heidelberg, Germany, 2007; pp. 1–28.
16. Georgiou, K.; Chamski, Z.; Amaya Garcia, A.; May, D.; Eder, K. Lost In Translation: Exposing Hidden Compiler Optimization Opportunities. *Comput. J.* **2020**, *65*, 718–735. [CrossRef]
17. Ng, B.; Prakash, A. Expose: Discovering Potential Binary Code Re-use. In Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference, Kyoto, Japan, 22–26 July 2013; pp. 492–501.

18. David, Y.; Yahav, E. Tracelet-Based Code Search in Executables. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, 16–21 November 2014; pp. 349–360.
19. David, Y.; Partush, N.; Yahav, E. Statistical Similarity of Binaries. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, 13–17 June 2016; pp. 266–280.
20. David, Y.; Partush, N.; Yahav, E. Similarity of Binaries through Re-Optimization. In Proceedings of the Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, 18–23 June 2017; pp. 79–94.
21. Yang, S.; Shi, Z.; Zhang, G.; Li, M.; Ma, Y.; Sun, L. Understand Code Style: Efficient CNN-based Compiler Optimization Recognition System. In Proceedings of the ICC 2019-2019 IEEE International Conference on Communications (ICC), Shanghai, China, 20–24 May 2019.
22. Egele, M.; Woo, M.; Chapman, P.; Brumley, D. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USA, 20–22 August 2014; pp. 303–317.
23. Sharma, R.; Schkufza, E.; Churchill, B.; Aiken, A. Data-Driven Equivalence Checking. In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, New York, NY, USA, 2013 ; OOPSLA '13, pp. 391–406.
24. Xu, X.; Feng, S.; Ye, Y.; Shen, G.; Su, Z.; Cheng, S.; Tao, G.; Shi, Q.; Zhang, Z.; Zhang, X. Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, Seattle, WA, USA, 17–21 July 2023.
25. Yang, S.; Xu, Z.; Xiao, Y.; Lang, Z.; Tang, W.; Liu, Y.; Shi, Z.; Li, H.; Sun, L. Towards Practical Binary Code Similarity Detection: Vulnerability Verification via Patch Semantic Analysis. *ACM Trans. Softw. Eng. Methodol.* **2023**, *32*, 1–29. [CrossRef]
26. Ramos, D.; Engler, D. Practical, Low-Effort Equivalence Verification of Real Code. In Proceedings of the Computer Aided Verification, Snowbird, UT, USA, 14–20 July 2011; Gopalakrishnan, G., Qadeer, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 669–685.
27. Jacobson, E.; Rosenblum, N.; Miller, B. Labeling Library Functions in Stripped Binaries. In Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools, New York, NY, USA, 5–9 September 2011; pp. 1–8.
28. Kruegel, C.; Kirda, E.; Mutz, D.; Robertson, W.; Vigna, G. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the Recent Advances in Intrusion Detection, Seattle, WA, USA, 7–9 September 2005*; Valdes, A., Zamboni, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; pp. 207–226.
29. Jang, J.; Woo, M.; Brumley, D. Towards Automatic Software Lineage Inference. In Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13), Washington, DC, USA, 14–16 August 2013; pp. 81–96.
30. Khoo, W.M.; Mycroft, A.; Anderson, R. Rendezvous: A search engine for binary code. In Proceedings of the 2013 10th Working Conference on Mining Software Repositories (MSR), San Francisco, CA, USA, 18–19 May 2013; pp. 329–338.
31. Partush, N.; Yahav, E. Abstract Semantic Differencing for Numerical Programs. In *International Static Analysis*; Logozzo, F., Fähndrich, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; pp. 238–258.
32. Partush, N.; Yahav, E. Abstract semantic differencing via speculative correlation. *ACM SIGPLAN Not.* **2014**, *49*, 811–828. [CrossRef]
33. Lahiri, S.; Hawblitzel, C.; Kawaguchi, M.; Rebêlo, H. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Proceedings of the Computer Aided Verification, Berkeley, CA, USA, 7–13 July 2012*; Madhusudan, P., Seshia, S.A., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 712–717.
34. Smith, R.; Horwitz, S. Detecting and Measuring Similarity in Code Clones. In Proceedings of the International Workshop on Software Clones (IWSC), Honolulu, HI, USA, 23 May 2009.
35. Lahiri, S.K.; Sinha, R.; Hawblitzel, C. Automatic Rootcausing for Program Equivalence Failures in Binaries. In *Proceedings of the Computer Aided Verification, San Francisco, CA, USA, 18–24 July 2015*; Kroening, D., Păsăreanu, C.S., Eds.; Springer International Publishing: Cham, Switzerland, 2015; pp. 362–379.
36. Hawblitzel, C.; Lahiri, S.K.; Pawar, K.; Hashmi, H.; Gokbulut, S.; Fern, o L.; Detlefs, D.; Wadsworth, S. Will You Still Compile Me Tomorrow? Static Cross-Version Compiler Validation. In Proceedings of the Foundations of Software Engineering (FSE'13), Foundations of Software Engineering (FSE'13) ed. ACM, Seattle, WA, USA, 13–18 November 2013.
37. Chen, J.; Patra, J.; Pradel, M.; Xiong, Y.; Zhang, H.; Hao, D.; Zhang, L. A Survey of Compiler Testing. *Acm Comput. Surv. CSUR* **2020**, *53*, 1–36. [CrossRef]
38. Daly, C.; Horgan, J. A Technique for Detecting Plagiarism in Computer Code. *Comput. J.* **2005**, *48*, 662–666. [CrossRef]
39. Duric, Z.; Gasevic, D. A Source Code Similarity System for Plagiarism Detection. *Comput. J.* **2012**, *56*, 70–86. [CrossRef]
40. Ragkhitwetsagul, C.; Krinke, J.; Clark, D. A comparison of code similarity analysers. *Empir. Softw. Eng.* **2017**, *23*, 2464–2519. [CrossRef]
41. Yuan, B.; Wang, J.; Fang, Z.; Qi, L. A New Software Birthmark based on Weight Sequences of Dynamic Control Flow Graph for Plagiarism Detection. *Comput. J.* **2018**, *61*, 1202–1215. [CrossRef]
42. Moussiades, L.; Vakali, A. PDetect: A Clustering Approach for Detecting Plagiarism in Source Code Datasets. *Comput. J.* **2005**, *48*, 651–661. [CrossRef]

43. Yang, J.; Hotta, K.; Higo, Y.; Igaki, H.; Kusumoto, S. Classification model for code clones based on machine learning. *Empir. Softw. Eng.* **2014**, *20*, 1095–1125. [CrossRef]
44. Alsubaihin, A.; Sarro, F.; Black, S.; Capra, L. Empirical comparison of text-based mobile apps similarity measurement techniques. *Empir. Softw. Eng.* **2019**, *24*, 3290–3315. [CrossRef]
45. Kaur, M.; Rattan, D. A systematic literature review on the use of machine learning in code clone research. *Comput. Sci. Rev.* **2023**, *47*, 100528. [CrossRef]
46. Mezouar, H.; El Afia, A. A Systematic Literature Review of Machine Learning Applications in Software Engineering. In *Proceedings of the 5th International Conference on Big Data and Internet of Things*; Springer: Cham, Switzerland, 2022; pp. 317–331.
47. Yang, Y.; Xia, X.; Lo, D.; Bi, T.; Grundy, J.; Yang, X. Predictive Models in Software Engineering: Challenges and Opportunities. *ACM Trans. Softw. Eng. Methodol.* **2022**, *31*, 1–72. [CrossRef]
48. Schkufza, E.; Sharma, R.; Aiken, A. Stochastic Superoptimization. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA, 14–18 March 2013; pp. 305–316.
49. Nain, M.; Goyal, N.; Rani, S.; Popli, R.; Kansal, I.; Kaur, P. Hybrid optimization for fault-tolerant and accurate localization in mobility assisted underwater wireless sensor networks. *Int. J. Commun. Syst.* **2022**, *35*, e5320. [CrossRef]
50. Ragkhitwetsagul, C.; Krinke, J. Siamese: Scalable and incremental code clone search via multiple code representations. *Empir. Softw. Eng.* **2019**, *24*, 2236–2284. [CrossRef]


**Disclaimer/Publisher's Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.