BINO: Automatic Recognition of Inline Binary Functions from Template Classes

Lorenzo Binosi[a,*], Mario Polino[a], Michele Carminati[a], Stefano Zanero[a]

[a]*Politecnico di Milano - Department of Electronics, Information and Bioengineering, Via Giuseppe Ponzio, 34, Milan, 20133, Italy, Italy*

Abstract

In this paper, we propose BINO, a static analysis approach that relieves reverse engineers from the challenging task of recognizing library functions that have been inlined. BINO recognizes inline calls of methods of C++ template classes (even with unknown data types). We do this through a binary fingerprinting and matching approach. Our fingerprint model captures syntactic and semantic features of an assembly function, along with its Control-Flow Graph structure. Using these fingerprints and subgraph isomorphism, it recognizes inline method calls in a target binary. BINO automates the fingerprints generation phase by parsing the source code of the template classes and automatically building appropriate binaries with representative inline calls of said methods. We evaluate BINO by performing experiments on a dataset of 555 GitHub C++ projects containing 9146 inlined functions, exploring several optimization levels that allow the compiler to inline function calls. We show that our approach can recognize inline function calls to the most used methods of well-known template classes with an F1-Score up to nuovo_valore % with the -O2, -O3 and -Ofast optimizations levels.

Keywords: reverse engineering, function inlining, template classes, function recognition, graph isomorphism

*Corresponding author

1. Introduction

Reverse engineering of binaries plays a key role in several security research areas, ranging from vulnerability discovery to malware analysis. Research on reverse engineering tools and techniques is thus of paramount importance. One of the most interesting open challenges is *binary function recognition*, that is, attempting to recognize a function as either a clone or a variation of another known function to ease the burden of the analyst and avoid reinventing the wheel. Unfortunately, this is not an easy task, as functions can be rewritten in a binary in different forms without changing their semantics: Instructions can be reordered, data types can be changed, and algorithms can be rearranged. In addition, when compilers translate high-level source code into machine code, there can be many possible outputs depending on optimization levels and other parameters. All of this without even considering voluntary obfuscation techniques, such as metamorphism in malware. *Function inlining* is a compiler optimization that replaces a `call` instruction with the body of the called function. When this happens, two or more functions effectively coexist in a single assembly function. This makes the recognition harder since syntactic and semantic analysis would need to consider that only a subgraph of the CFG is responsible for a function. Inlining happens with all sorts of functions, but it is widespread within template classes from the C++ standard library: Methods of such classes are the perfect candidates for inlining optimizations.

With this work, we propose BINO, an approach based on binary function fingerprinting to recognize inline method calls of template classes from the C++ standard library. Our fingerprints capture syntactic and semantic aspects of an assembly function, as well as the structure of its CFG. Fingerprints can be automatically generated (from the parsing of the C++ standard library to the creation of the fingerprint database). Using subgraph algorithms, we can identify inline calls of fingerprinted functions in binary applications. We implement the proposed approach in a framework composed of several independent modules that we use to build a database of the most relevant methods of library template classes (i.e., fingerprints). Such a database can then be used to recognize the fingerprinted methods as inline methods in a target binary application.

We test BINO on a dataset of 555 C++ GitHub projects, compiled with different optimization levels. On the resulting binaries, we search for the fingerprints of methods of classes `std::map`, `std::vector`, and `std::deque`,

since they are the most used library classes. We then verify each match through DWARF debugging information [1]. BINO achieves F1-Scores of 65%, 64%, and 41%, with the optimization levels `-O2`, `-O3/-Ofast`, and `-Os` respectively. In particular, when considering known template parameters (i.e., data types used for building the fingerprints), BINO achieves an Hit-Rate up to 77%. While, with unknown template parameters, our approach accomplishes an Hit-Rate up to 51%. This demonstrates how the fingerprints generated by BINO can generalize by capturing the most relevant features of the inline methods. Moreover, we also show how BINO can attain a precision of 100% when considering the `operator[]` of the `std::map` class, which is the most complex method under analysis. This suggests how the precision in the recognition scales with the complexity of the method under analysis.

In summary, the main contributions of this work are as follows:

- We propose BINO, an approach to recognize inline method calls of template classes. To the best of our knowledge, this is the first approach to inline function recognition. Indeed, other approaches aim to recognize entire functions.
- We propose a fingerprint that captures syntactic and semantic aspects of assembly functions for matching inline code.
- We develop a binary fingerprinting and matching framework able to generate relevant fingerprints from the source code of a template class. We make the source code of the framework publicly available[1], both for the reproducibility of our experiments and to encourage further research.
- We evaluate BINO on a representative set of C++ projects[2], compiled with different optimization levels, against the template classes `std::map`, `std::vector`, and `std::deque`.

2. Background and Motivation

Function recognition is one of the core activities in modern static analysis. It is very usefull in many applications ranging from control flow integrity to

[1] https://github.com/necst/BINO
[2] https://github.com/necst/BINO_Dataset

binary similarity and vulnerability detection. Therefore, it is implemented in many binary analysis tools (e.g, Angr [2], Ghidra, IDA, rev.ng [3], BAP, Radare, Binary Ninja, Hopper, Objdump), since detecting a binary function provides the core functionality to understand and analyze the high-level semantic of a low-level binary [4]. While it is easy to understand the semantics of a function when you have symbols or debugging information, it becomes drastically challenging when the function is stripped or compiler optimizations are applied. This is particularly relevant for the task of vulnerability detection where reverse engineers need to identify functions and their semantics in the application code. However, code obfuscation techniques and compiler optimizations, such as inlining, make state-of-the-art tools not up to the challenge.

2.1. Inlining

Modern compilers use multiple methods to optimize emitted code toward different metrics (e.g., the binary size or the runtime performance). Since discussing this in-depth is beyond the scope of this paper, you can read [5] for an overview of the subject.

One of the techniques used for optimization by compilers is *function call inlining*, i.e., substituting a function call with the body of the function itself. There are several reasons why this can be beneficial. First, the `call` instruction itself is expensive to execute. Also, if the function's code is tiny, the prologue and epilogue can significantly – and pointlessly – add to its size. Finally, inlined code can be further optimized along with surrounding code, something not possible otherwise. For these reasons alone, inlined code is generally faster.

On the other hand, inlining has its drawbacks and may lead to worse performance. For instance, the presence of multiple copies of the same function code across the binary can lead to instruction cache misses [6]. Indeed, the duplicated code will be loaded several times into the cache, leading to inefficient cache usage. In addition, inlining increase the overall binary size and negatively impacts loading time, as well as potentially causing issues on memory-constrained architectures.

For these reasons, during optimization, a set of heuristics evaluates whether or not to inline a specific function call, taking into account parameters such as the number of times the function is called, its size, and its execution time.

2.2. Problem Statement

Inlining makes the task of reverse engineering significantly more complex and time-consuming. Indeed, as a reverse engineer, the only way of recognizing inlined functions is to spend significant time understanding the code's semantics while reversing complex functions. Current approaches [7, 8, 9] focus only on identifying assembly functions as clones of known ones, possibly library functions. However, when inlining is in place, it can drastically change the code of assembly functions. For this reason, their analysis also considers the assembly code of the internal callees. Despite solving an important problem, state-of-the-art approaches do not explore the more challenging problem of detecting inlined library functions, which plays an essential role in the reversing task.

Inlined library functions are widespread in modern C++ applications due to the broad utilization of standard library template classes (a.k.a. containers) such as `std::vector` and `std::map`. Whenever C++ applications use methods from template classes, the compiler emits ad-hoc assembly code to deal with the chosen template parameter(s). Hence, the emitted assembly code is part of the binary application since it is only relevant to the application itself. In addition, methods from template classes are the best candidates for inlining. Therefore, such methods are likely inlined in the final binary application, especially if compiler optimizations are in place. Ultimately, the resulting assembly code contains possibly complex inlined library code mixed up with user-defined code.

To better understand the goal and challenges in recognizing such inline binary functions, we refer to the example in Listing 1 for the rest of this section. In the code, the function `add_to_vector` asks for an integer from the command line, squares it, and appends the result to a `std::vector` through the `push_back` method. The Control-Flow Graphs (CFGs) of the function compiled with optimization levels `-O0` and `-O3` are reported in Figure 1a and Figure 1b respectively. The instructions in red highlight the main differences due to inlining: on the left, the compiler emits a function call to the `push_back` method, which we can see as the only instruction highlighted in red, while on the right, the compiler inlines such a call and thus, the instructions highlighted in red are the ones of the `push_back` method. Moreover, we can observe the effect of inlining on the rightmost CFG due to the call to the function `std::vector::_M_realloc_insert`, an internal and private method called by the function `push_back`. This example mainly describes the ultimate goal of recognizing inline binary functions, i.e., highlighting the in-

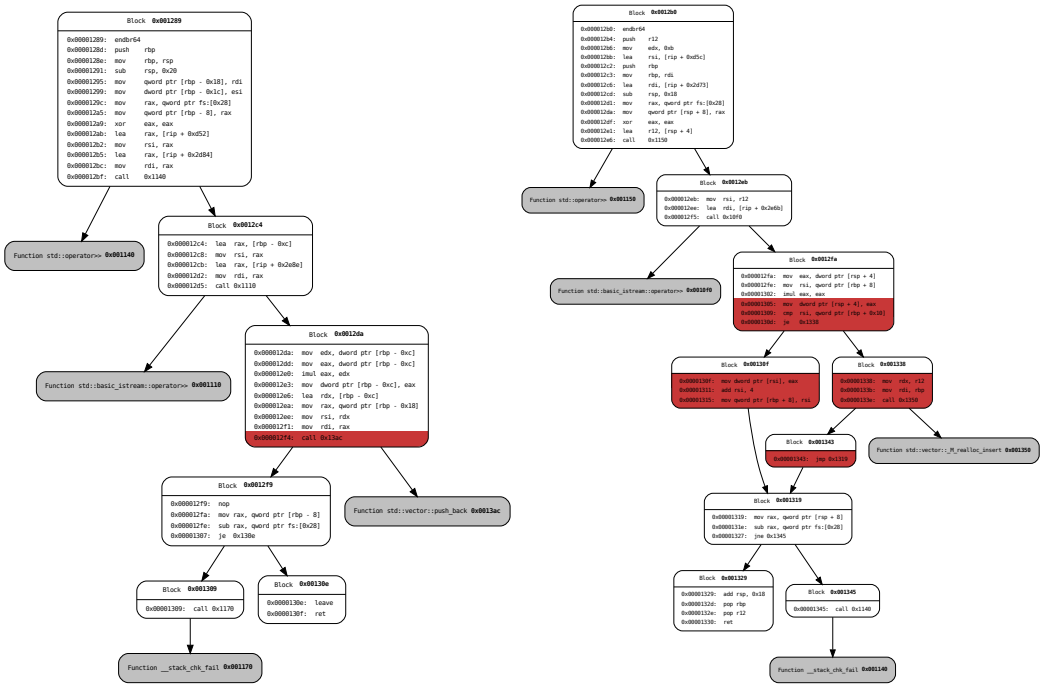
```
1  void add_to_vector(std::vector<int> vec) {
2    int val;
3
4    std::cout << "New value: ";
5    std::cin >> val;
6    val = val * val;
7    vec.push_back(val);
8  }
```

Listing 1: C++ template example.

structions that belong to an inline binary function. Moreover, it also gives an intuition about the challenges of this task, which we discuss in the following.

Multilevel Inlining. *Multilevel inlining* is the process of inlining function calls recursively. For instance, function `foo` may call function `bar` which in turn may call function `baz`. In such a case, the compiler may decide, according to some heuristics and considering several aspects of these functions (e.g., their size, execution time, etc.), to inline both the function calls to `bar` and `baz`. This is called multilevel inlining, and this is an instance of 2-level inlining. In the example in Figure 1b, we have instances of multilevel inlining as well, but it is not easy to spot since we only have the compiler output. Moreover, the example we propose is only an instance of the compiler output. In another one, the compiler may inline the `_M_realloc_insert` private method call and, possibly, some of the functions called by this latter, increasing the amount of inlined code. Thus, understanding the code semantics when the amount of code changes across binaries, and possibly even across functions of the same binary, is a challenging task.

Template Parameters. Template classes are a *metaprogramming* feature where C++ code is generated according to programmer-specified template parameters: for instance, we can create a `std::vector` object containing `int` elements; but the vector could also contain objects of a different standard class (e.g., `std::string`), or even objects created with a custom class. Moreover, depending on the chosen template parameter, we have different assembly codes. Indeed, template classes methods (e.g., `std::vector::-push_back`, `std::map::operator[]`) call methods of the template parameter (e.g., the copy constructor or the destructor) to perform operations on the objects inside the container. When multilevel inlining reaches these function calls from the method we want to recognize, the assembly code will



(a) Compiled with `-O0`.

(b) Compiled with `-O3`.

Figure 1: CFGs of the `add_to_vector` function in Listing 1 compiled with `-O3` (1b) and `-O0` (1a).

also include the template parameter(s) code. For instance, if we have a `std::vector` of `std::string`, the inlined code emitted for the `push_back` of such vector might include the inlined code of the copy constructor of the `std::string`.

Constant Folding - Constant propagation - Dead Code Elimination. When the compiler decides to inline a function call, it replaces the body of the callee with the function call. By doing this, it can consider the code and the variables of the callee as code and variables of the caller. Moreover, it can apply further optimizations such as *Constant folding*, *Constant propagation*, and *Dead code elimination*. Thanks to these optimizations, the compiler makes the semantics of the inlined functions less generic whenever constant values are in place at compile time. For instance, it can cut an entire branch of an inlined function when it knows the result of an `if` statement: something that could not be possible if the function is not inline, as it must preserve the semantics of the original function.

Initial and Final Instructions. To make things even more challenging, inline functions are often optimized along with the code of their (often complex) caller functions. This produces basic blocks that can contain a mixture of instructions from both the caller and the callee, located at both the beginning and the end of the inlined code. Thus, to recognize inline functions, we need to consider that the initial and final basic blocks may contain instructions unrelated to the function that we are trying to identify. We can see this behavior once again in Figure 1b, where the initial basic block of the inlined method contains some instructions of the caller, i.e., the ones that perform the squaring, and some instructions of the callee, i.e., the initial instructions of the `push_back`.

In summary, although the problems of binary similarity and function identification have been of interest for a long time in the research community, identifying inline functions is an even harder and not fully explored research challenge. The problem is of high practical relevance since inlining is very common and, at the same time, challenging for reverse engineers. In particular, the inlining of library functions is needlessly time-consuming, making reversers spend time analyzing well-known code. This often happens for functions that are methods of template classes and, in particular, those from the C++ standard library due to their inherent characteristics. Automatic recognition of these functions is, thus, extremely challenging and of utmost practical importance.

For these reasons, we develop a novel approach to perform such recognition and develop BINO to simplify this complex task.

3. Identification of Inline Functions

Our approach, named BINO, aims at recognizing inline methods from C++ template classes in binaries. BINO can automatically extract relevant features from the assembly methods we want to recognize as inline methods in a target binary. Moreover, it automates the generation of the assembly methods themselves, making the overall features extraction process completely automatic. The extracted features are combined in a *fingerprint* used for the matching. In Figure 2, we show a high-level overview of the approach, which is composed of six modules. The first five modules take care of the generation of the fingerprint database, starting from the source code of the class(es) we want to recognize. The final module performs the matching and the identification of the inline methods.

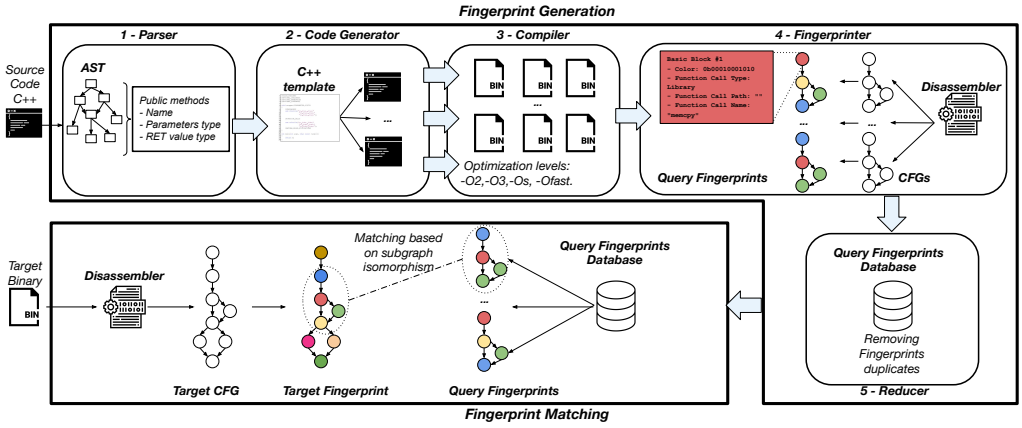


Figure 2: High-level overview of the BINO framework architecture.

3.1. Fingerprint Generation

Our fingerprints are based on the CFG of the assembly method. The fingerprints are enriched with two features that capture their syntactic and semantic aspects: ❶ The *mnemonic groups* (i.e., colors) of the assembly instructions contained in each basic block of the method, and ❷ information about the function calls in each basic block.

We divide the assembly instructions in *mnemonic groups* [10, 11, 12] of closely related operations. For each basic block, we represent mnemonic groups as a bit vector. When an instruction of a giving group is present in the basic block, the bit associated with the group is set to one. We call this bit vector *color*, in analogy with the RGB data representation (and with [11, 12]). Within a fingerprint, we also annotate each basic block with information about the function calls. They are the name of the function called, and the type of function call. All the details about mnemonic groups and function call information are reported in Appendix A.

In order to build our fingerprints, we first need to compile and disassemble binaries in which these methods are called. The overall process of fingerprint generation is described in the topmost part of Figure 2.

Parser. The first module parses the source code of the template class(es) we want to fingerprint. In order to correctly parse the source code, we build the Abstract Syntax Tree (AST) using *clang-cindex* python package, which provides a tool able to parse C/C++ source code. By inspecting the AST, we are then able to extract the information about the public methods of the

```
1  void wrapper($PARAMETER_LIST$) {
2      $OPTIONAL_VARIABLES$
3      asm volatile("or␣%rax,␣%rax;"
4                   "or␣%rax,␣%rax;"
5                   "or␣%rax,␣%rax;");
6      $FUNCTION_CALL$
7      asm volatile("or␣%rbx,␣%rbx;"
8                   "or␣%rbx,␣%rbx;"
9                   "or␣%rbx,␣%rbx;");
10     $OPTIONAL_RETURN_VALUE_UTILIZATION$
11 }
```

Listing 2: C++ tokenized source code.

class(es). In particular, we need the name of the methods, their parameters data types, and their return value data types. The extracted information is passed as input to the *Code Generator*.

Code Generator. The purpose of this module is to generate code samples of inline methods that can be used as input by the compiler module to generate binary samples for the fingerprint generation process. Besides calling the method under analysis, this module takes care of three requirements: ❶ Ensure that the beginning and the end of our method are identifiable, ❷ avoid that optimization of the control flow eliminates part of the method, ❸ produce samples code for all the data types. To achieve such requirements, this module uses C++ tokenized code similar to the one in Listing 2.

We insert a few *marker* assembly instructions in the code (using the `asm` function) in order to understand where the inline method starts and ends (Requirement ❶). Such instructions do not change the values in the registers and cannot be optimized by the compiler because of the `volatile` keyword. Requirement ❷ is the most challenging one. Indeed, if the compiler can infer the state of the object at compile time, it can remove part of the code of the inline method that will never be executed. To avoid this – and generate a complete fingerprint – we employ a few tricks in the generated code. For example, we enforce the declaration of the object outside the wrapper function, and we ensure that the return value of the inline method is used by the wrapper function. For example, if we want to call the `push_back` method of the `std::vector` class with an `int` template parameter, we need a `std::vector<int>` object and an `int` variable. We define those outside the wrapper function. Moreover, the module must build a source code instance for each (combination of) data type(s) used as a template parameter of the

target and for each method (Requirement ❸). For example, in order to fingerprint all of the public methods of `std::vector` with each of the following data types: `int`, `double`, `float`, `char`, `char*` and `std::string`, the code generator produce 6 different source code files for each public method of the class. As we will discuss in Section 5, we cannot do this for each possible data type. Our aim is thus to generate fingerprints that capture the most common scenarios. Although, the framework is built to easily extend the code generation with new data types. Finally, we repeat this procedure for all the tokenized source codes. We built tokenized source code to capture several CFG structures of the inline methods. For instance, we have a case in which the function call to the method is performed multiple times and cases in which the function call is within the body of an `if` or `for` statement.

Compiler. This module compiles all the C++ source code files generated by the previous module with three optimization levels: `-O2`, `-O3`, `-Os` and `-Ofast`. These are all the optimization levels provided by the GCC compiler that adopt inlining. Thus, the output of this step is one binary executable for each optimization level (i.e., three) per each source code sample produced by the code generator.

Fingerprinter. The task of this module is to extract the fingerprints of the methods from the previously compiled binaries. In order to extract such fingerprints, we first need to disassemble the binaries. For this task, we rely on *Angr* [2], a binary analysis framework for both static and dynamic symbolic analysis. We use it on each binary to extract, through static analysis, the assembly instructions, the function call information, and the CFG of the wrapper function. It is useful to note that the *Angr* disassembler splits a basic block after a function call. Thus, in the basic blocks of a fingerprint, we can have at most one function call in each block. Also, *Angr* produces all of the function call information we need in our annotation of the blocks.

We remove all the assembly instructions that are not enclosed between the *marker* instruction blocks we inserted in Code Generation (Section 3.1). At the end of this process, we are left with the CFG of the inline function. To build a fingerprint, we enrich the CFG with the features we have described at the beginning of Section 3.1. These features are the colors and the function call information. The color is stored as a bit vector for each basic block of the CFG where each bit represents one mnemonic group of Table A.7.

Reducer. The process we have just described generates a large dataset of fingerprints that often contain duplicates (e.g., often source code compiled

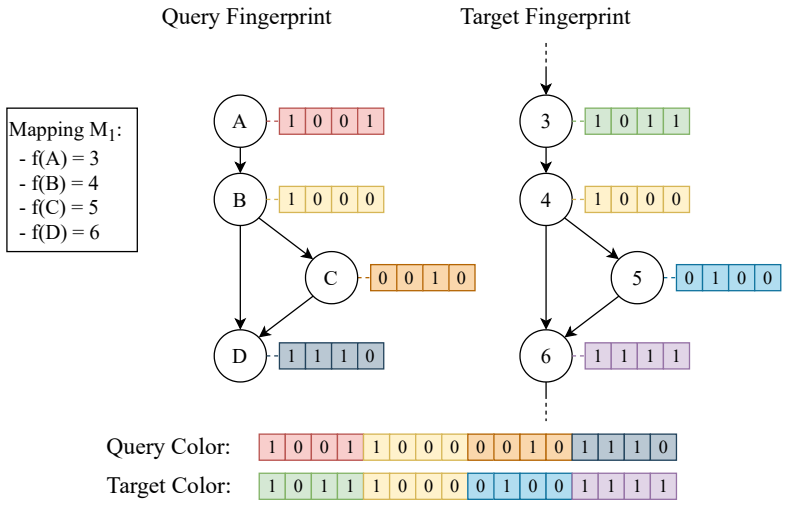


Figure 3: Example of the creation of a color bit vector for a query and a target fingerprint matching subgraphs.

with options `-O2` and `-O3` generates the same fingerprints). For this reason, the Reducer module seeks and removes duplicates from the dataset. First, we seek fingerprints with an isomorphic CFG structure [13, 14]. As a recall, two graphs G and H are isomorphic, if there exists a bijection $f : V_G \to V_H$ between vertices of the graphs such that if $\{a, b\}$ is an edge in G, then $\{f(a), f(b)\}$ is an edge in H. To find an isomorphism, we rely on *VF2* [15, 16], a graph isomorphism algorithm provided by the *NetworkX* python package. Once an isomorphism is found, we have one or more mappings between the basic blocks of the first CFG and the basic blocks of the second CFG. For each of these mappings, we finally check the color bit-vectors and the call information of each pair of basic blocks in the mapping. If colors and call information are equal for all the pairs of at least one of the mappings, we have a perfect duplicate, and we can discard one of the two fingerprints.

3.2. Fingerprint Matching

The last module of the framework performs the inline function recognition on an unseen binary (i.e., target binary). In particular, it finds the inline functions in the target binary by matching the fingerprints in the database built by the previous modules.

First, this module fingerprints all of the assembly functions in the binary, using the same process described in Section 3.1. We will refer to these fin-

gerprints as "target fingerprints", whereas the fingerprints from the query database will be called "query fingerprints".

Second, the framework searches for *subgraph isomorphisms* [17] between the CFGs of each of the query fingerprints and each of the target fingerprints. A graph G is subgraph isomorphic to a graph H, if there exists an injection $f : V_G \to V_H$ between vertices of the graphs such that if $\{a, b\}$ is an edge in G, then $\{f(a), f(b)\}$ is an edge in H. In other words, a graph G is subgraph isomorphic to a graph H, if there exists a graph isomorphism between G and a subgraph of H.

To avoid generating a large number of non-substantial matches, we set a minimum number M of basic blocks of a query fingerprint as a sensitivity parameter. Moreover, depending on where the call to the method is performed and due to optimizations applied by the compiler, the generated CFG can change significantly. We handle these cases by changing the CFGs of our query fingerprints before performing the subgraph isomorphism algorithm. The details of this operation are reported in Appendix B.

For each subgraph isomorphism found, we can have more mappings between the basic blocks of a query fingerprint and the basic blocks of a target fingerprint. This can happen when a method is called and inlined multiple times in another function. Hence, for each of these mapping, we check if the corresponding pairs of basic blocks have the same function call information (i.e., function names and types). It is important to note that we may not always be able to compare function names (e.g., if a function is part of a stripped binary). In this case, we ignore the name of the functions, comparing just the function call types. We will always be able to obtain and compare the symbols of library functions since they are required by the binary application for their correct functioning.

We finally compare the color information for each subgraph isomorphism found where the function call information matches. To compare the colors, we first deal with the extra instructions of the caller that can be present in the initial and final(s) basic blocks of the target fingerprint. In fact, during inlining, the first basic block of the callee is combined with the basic block of the caller, and similarly, the last basic block of the callee is fused with the landing basic block that follows the function call. In order to make these instructions not affect our analysis, we simply remove the mnemonic groups (i.e., colors) that are not active in the corresponding basic blocks of the query fingerprint. For instance, if the initial basic block of the match (target) has the bit relative to the floating-point operations set, while the corresponding

query basic block does not, we set the bit to zero in the target basic block. Then, we compute the Jaccard coefficient: $J(Q,T) = |Q \cap T|/|Q \cup T|$ between all the basic blocks' mnemonic groups (i.e., colors) and discard matches where the coefficient is lower than a specified sensitivity parameter S. This process is illustrated through an example in Figure 3. The final output is a list of the inline methods recognized and the binary addresses of the basic blocks that belong to the recognized method.

4. Evaluation

The main goals of this experimental evaluation are:

1. Prove that our approach can recognize inline methods from the C++ template classes.
2. Prove that our approach can generalize by recognizing inline functions of unknown data types.
3. Study the influence of function size on the performance.
4. Analyze the impact of syntactic and semantic features used to enrich the CFG.

We run all of the experiments in an `Ubuntu 18.04` docker container with `g++` version `7.5.0`. All compilations are performed with `x86_64` as target architecture and the `C++14` language version.

4.1. Query Fingerprints Database

We choose to fingerprint the methods of the template classes `std::map`, `std::vector` and `std::deque` for our experiments. They are the most known and used C++ template classes and they are available in many online projects. They are also extremely generic and, therefore, particularly challenging to detect when inlined. Nevertheless, we show that our approach can correctly identify those methods from template classes.

We configure the Code Generator (see Section 3.1) to use a set of standard data types as template parameters. `std::vector` and `std::deque` need only one data type as value parameter, whereas the `std::map` has two template parameters (key type, value type). All the data types used are described in Table 1. We generate fingerprints for all the combinations of those data types. As described in Sectiong 3.1 all methods of these classes, for all combinations of parameters, are compiled against each of the following optimization levels: `-O2`, `-O3`, `-Os`, and `-Ofast`. With this configuration, we extract fingerprints (details in Section 3.1) that are part of our test database as described.

Table 1: Data types used as parameters.

Template	Parameter	Data Type
`std::map`	Key Type	`int`, `long long`, `std::string`
	Value Type	`int`, `double`, `float`, `char`, `char*`, `std::string`
`std::vector`	Value Type	`int`, `double`, `float`, `char`, `char*`, `std::string`
`std::deque`	Value Type	`int`, `double`, `float`, `char`, `char*`, `std::string`

4.2. Dataset Selection

To create a dataset for an automated validation phase, we consider all of the GitHub projects tagged as `C++` and `Makefile`. We find approximately 50,000 such projects. Then, we download all of them and discard the projects that cannot be automatically compiled, as well as the ones with compilation errors. We also discard the ones in which we cannot automatically change the compilation options project-wide using the `CXXFLAGS` variable. At this point, we are left with approximately 2,000 filtered projects.

To make our experiment meaningful, we extract from this set the projects that use the template classes for which we have generated fingerprints. Our final dataset includes 555 projects, 500 of them using the `std::vector` class, 217 using the `std::map` class, and 57 using the `std::deque` class (the three sets, of course, are not disjoint). To ensure the reproducibility of our experimental results, we intend to make the dataset openly accessible to the scientific community. This will allow other researchers to access the data and conduct their own analyses, potentially leading to further advancements in the field.

Furthermore, we split the dataset into two equal parts. We use the first one to valide the parameters S and M, and the second one to test the performance of our framework.

The projects in the final dataset are compiled in the same environment described above and with the optimization levels `-O2`, `-O3`, `-Os`, and `-Ofast`. Moreover, they are compiled with DWARF debugging information (`-g` option). DWARF [1] is a debugging file format used by many compilers and debuggers to support source-level debugging. You may notice that this also necessarily includes the symbols in the binary, which of course, we ignored while performing our tests and used only in the validation.

We parse the DWARF debugging information looking for `DW_TAG_inlined_subroutine` tags to build the ground truth. These tags tell us the mangled names of the inline functions and their memory ranges. Each time we have a match, we verify its correctness through this information. However, note that even if a method has generated fingerprints, and thus it has at least M basic blocks in its CFG, there is no guarantee that the same method, compiled within a target binary, will also be composed of M or more basic blocks. This is mainly caused by internal functions that are not inlined. In fact, internal calls can either be inlined or not. Hence, if an inline method in a target binary is composed of less than M (in our case, 5) basic blocks, we exclude its recognition from our experiment. To check if an inline method in a target binary is at least of M basic blocks, we disassemble the target binary with *Angr*. With the DWARF debugging information, we retrieve the memory range(s) of the inline method, and we map such memory addresses into basic blocks disassembled by *Angr*. However, it should be noted that DWARF debugging information may not always be entirely reliable. There may be instances where instructions or basic blocks are not accurately marked as part of an inline function, even when they are. Consequently, this may introduce noise in the ground truth, which could potentially hinder the recognition process. Nonetheless, it is important to note that such occurrences are sporadic and should have minimal negative impact on the overall performance of BINO.

4.3. BINO Metrics

To evaluate the performance of BINO, we use the metrics that are commonly used in state-of-the-art assembly clone identification studies. These metrics are Precision, Recall and F1-Score. We denote with N the number of times the compiler has inlined each method. We denote with TPs (true positives) the number of inline methods that we recognize correctly. Furthermore, we denote with FPs (false positives) the number of recognitions that do not match our ground truth. Moreover, we compute three standard metrics, the Precision $Prec = \frac{TPs}{TPs+FPs}$, the Recall $Rec = \frac{TPs}{N}$, and the F1-Score $F1 = 2 \times \frac{Prec \times Rec}{Prec+Rec}$.

4.4. Parameters Selection

Before testing the performance of our framework, we need to select proper values for the parameters S and M. To do so, we use the first half of our dataset to select them. Figure 4 shows the F1-Score of BINO depending on

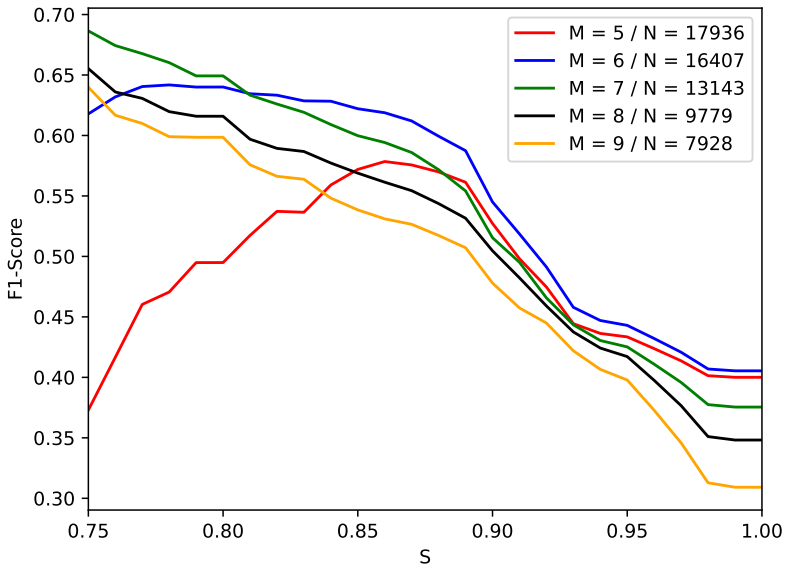
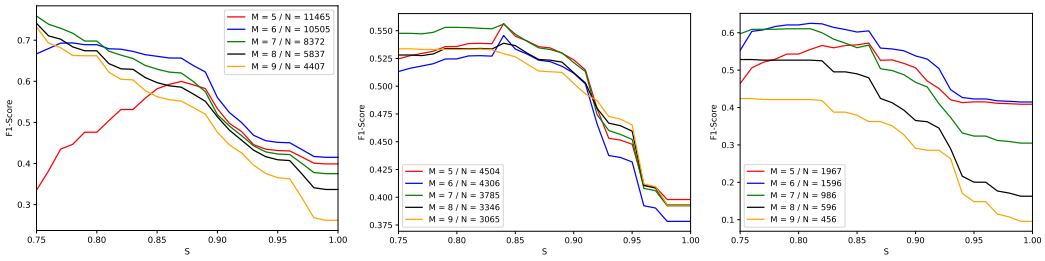


Figure 4: BINO overall F1-Score depending on S and M.



(a) std::vector (b) std::map (c) std::deque

Figure 5: BINO F1-Score depending on S, M and the class under test.

the value of S and M. As you can see, the number of samples, i.e. inlined functions in our dataset, decreases as M increases. This is expected since we do not consider all the samples that are composed by less than M basic blocks. However, as you can see, choosing M becomes a trade-off between BINO performance and usability (i.e., number of inlined functions we can identify). For this reason, we pick $M = 6$ to keep a broad number of samples and we pick $S = 0.78$ since it maximizes the F1-Score.

Given that our dataset is unbalanced, we want to provide the values of S and M depending on the class we want to recognize. Figure 5 shows the F1-Score of BINO depending on the value of S and M for each class of the test

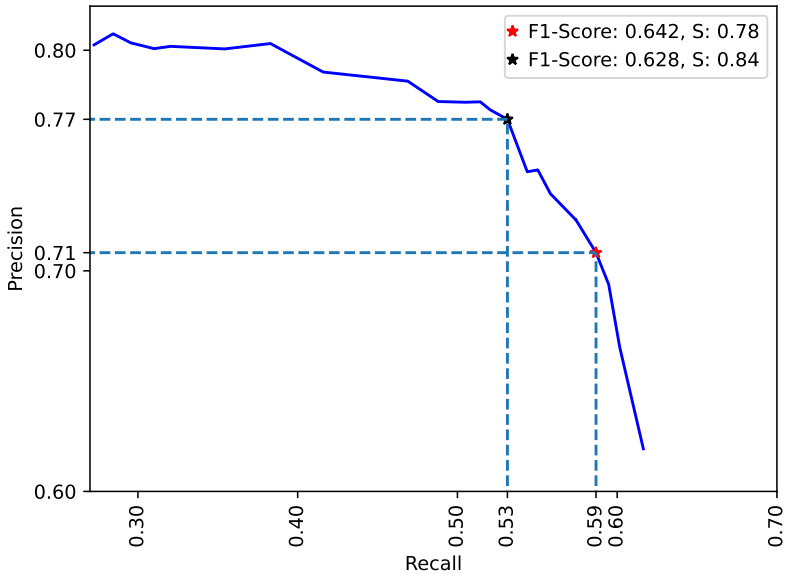


Figure 6: Precision-recall curve

set. As you can see the value of S and M slighty differs for each class under test. For the class `std::vector` (Figure 5a), the best values are $M = 6$ and $S = 0.78$. Instead, for the class `std::map` (Figure 5b), the best values are $M = 5$ and $S = 0.84$. Finally, for the class `std::deque` (Figure 5c), the best values are $M = 6$ and $S = 0.81$.

Finally, in Figure 6, we provide an overview of the precision-recall curve, which demonstrates the performance tradeoff considering the parameter S. We compute the precision-recall using the value of M found previously, i.e., $M = 6$. We can identify two main operating points: ❶ identified by the elbow point (black mark), and ❷ identified by the maximum value of the F1-Score (red mark). It is important to note that ❷ is the same operating point identified above.

4.5. BINO Matching Performance

In this experiment, we evaluate the performance of BINO in recognizing inlined functions from the C++ standard library using the second half of our dataset. As mentioned in Section 4.4, we set $M = 6$ and $S = 0.78$. The results of our experiment are reported per method and per optimization level in Table 2.

As shown in Table 2, BINO achieves a precision of 72% and a recall of

Table 2: Overall results per method.

Function Name		std::map::operator[]	std::map::lower_bound	std::map::upper_bound	std::map::find	std::map::erase	std::map::at	std::vector::push_back	std::vector::resize	std::vector::clear	std::vector::reserve	std::vector::erase	std::vector::insert	std::vector::emplace_back	std::deque::push_back	std::deque::pop_back	std::deque::push_front	std::deque::pop_front	std::deque::operator[]	Overall
–O2	**N**	641	756	43	201	6	27	1671	183	131	36	51	13	330	101	3	6	25	102	4326
	TPs	274	416	14	17	4	11	1122	158	58	1	5	4	163	80	2	5	24	81	2439
	FPs	41	89	53	2	29	0	213	107	32	0	6	0	137	127	0	4	11	92	943
	Prec	0.87	0.82	0.21	0.89	0.12	1.00	0.84	0.6	0.64	1	0.45	1.00	0.54	0.39	1.00	0.56	0.69	0.47	0.72
	Rec	0.43	0.55	0.33	0.08	0.67	0.41	0.67	0.86	0.44	0.03	0.1	0.31	0.49	0.79	0.67	0.83	0.96	0.79	0.56
	F1	0.57	0.66	0.25	0.66	0.21	0.58	0.75	0.71	0.52	0.05	0.16	0.47	0.52	0.52	0.80	0.67	0.80	0.59	0.63
–O3	**N**	544	654	36	274	15	30	1972	196	122	64	38	24	727	124	3	17	26	78	4944
	TPs	200	371	16	28	13	10	1359	154	57	1	6	6	372	70	0	8	23	63	2757
	FPs	0	115	74	86	91	0	241	131	19	0	6	0	165	95	14	6	5	43	1091
	Prec	1	0.76	0.18	0.25	0.12	1.00	0.85	0.54	0.75	1	0.50	1.00	0.69	0.42	0.00	0.57	0.82	0.59	0.72
	Rec	0.37	0.57	0.44	0.10	0.87	0.33	0.69	0.79	0.47	0.02	0.16	0.25	0.51	0.56	0.00	0.47	0.88	0.81	0.56
	F1	0.54	0.65	0.25	0.14	0.22	0.50	0.76	0.64	0.58	0.03	0.24	0.40	0.59	0.48	0.00	0.52	0.85	0.68	0.63
–Os	**N**	68	92	21	80	6	3	340	75	1	37	22	13	160	28	–	1	2	2	951
	TPs	14	19	7	11	5	0	211	22	1	1	5	5	104	19	–	0	1	0	425
	FPs	0	56	56	149	15	0	10	138	1	0	1	0	7	15	–	3	0	10	461
	Prec	0.21	0.21	0.33	0.14	0.25	0.00	0.62	0.29	1.00	0.03	0.23	1.00	0.94	0.68	0.00	0.00	0.50	0.00	0.48
	Rec	1.00	0.25	0.11	0.07	0.83	0.00	0.95	0.14	0.50	1.00	0.83	0.38	0.65	0.56	0.00	0.00	1.00	0.00	0.45
	F1	0.34	0.23	0.17	0.09	0.38	0.00	0.75	0.19	0.67	0.05	0.36	0.56	0.77	0.61	0.00	0.00	0.67	0.00	0.46
–Ofast	**N**	544	654	36	274	15	30	1955	201	122	63	38	24	716	124	3	17	26	78	4920
	TPs	200	371	16	28	13	10	1344	160	57	1	6	6	368	70	0	8	23	63	2744
	FPs	0	115	74	90	91	0	241	132	19	0	6	0	165	95	14	6	5	43	1096
	Prec	1	0.76	0.18	0.24	0.12	1.00	0.85	0.55	0.75	1	0.50	1.00	0.69	0.42	0.00	0.57	0.82	0.59	0.71
	Rec	0.37	0.57	0.44	0.10	0.87	0.33	0.69	0.80	0.47	0.02	0.16	0.25	0.51	0.56	0.00	0.47	0.88	0.81	0.56
	F1	0.54	0.65	0.25	0.14	0.22	0.50	0.76	0.65	0.58	0.03	0.24	0.40	0.59	0.48	0.00	0.52	0.85	0.68	0.63

56% overall, with the most used optimization levels, i.e., `-O2-O3`, and `-Ofast`. These values are 47% and 45% respectively for the optimization level `-Os`.

Since BINO is, to the best of our knowledge, the first tool performing inline function recognition, we do not have a direct comparison with previous works. However, we can consider the results of cited state-of-the-art works [8, 10] performing the easier task of function recognition without inlining. Although these tools achieve slightly higher overall performance in the generic task of function recognition, with a recall of around 90%, our tool is capable of performing a much more challenging task with a comparable (if slightly lower) precision and recall.

If we consider the methods under analysis individually, we mostly obtain the best performance with the `-O2` optimization level and slightly lower ones with `-O3` and `-Ofast` optimization levels. This is caused by the additional optimizations performed by `-O3` and `-Ofast` that we cannot always capture. In fact, our way of generating code samples is relatively simple and sometimes cannot produce all the relevant cases the compiler may emit. Instead,

we obtain the worst performance with the `-Os` optimization level. This is again caused by our code generator, which is relatively simple and does not consider cases in which multiple methods of the same template class are used within the same source code. Most of the time, such methods share calls to private and template parameters methods that are not inlined when compiled with `-Os`. Indeed, when `-Os` is enabled, the compiler rarely inlines a method that is called twice. On the contrary, they are inlined in our fingerprints since we call the method we want to fingerprint only once. We achieve good performance with the most known and used methods of the template classes under analysis, namely `std::map::operator[]`, `std::map::lower_bound`, `std::deque::operator[]`, `std::deque::pop_front`, `std::deque::push_back`, `std::vector::resize`, and `std::vector::push_back`. These results highlight the value of the tool in supporting the reverse engineering process when considering real-world binary applications. The worst performance is obtained instead for the `std::vector::reserve` method. Such a method is used to reserve a certain amount of memory area (provided as a parameter) for the `std::vector` object. Most of the time, this method is called immediately after instantiating the vector class through the constructor. Thus, as discussed in Section 2, the assembly code of the `std::vector::reserve` is merged with the code of the constructor and simplified, leading to a few basic blocks that we are not able to recognize. However, we have a high precision because when the case we just described does not happen, the method is complex enough to generate a characteristic sequence of basic blocks.

4.6. Model Transferability

You could wonder what happens if the parameter(s) of the template classes whose methods are used and inlined in the target binary are not the same data types or classes that we have used and fingerprinted in building our query fingerprint database.

The goal of our framework is to capture the structure of methods and find the same structure even when there are modifications. Indeed, we can show that our framework can generalize and find correct matches against methods generated with different template parameters than the ones we used to build the query database.

To better analyze this phenomenon, we split the results depending on the template parameters used to create the objects of the classes. If an object is created using as a parameter a data type that we also used to create our fingerprints database, we call it a *known template parameter*. Otherwise, we

Table 3: Results per method with known template parameters.

Function Name	Known template parameters											
	–O2			–O3			–Os			–Ofast		
	Found	Hit	Rate	Found	Hit	Rate	Found	Hit	Rate	Found	Hit	Rate
std::map::operator[]	171	69	0.4	78	29	0.37	15	6	0.4	78	29	0.37
std::map::lower_bound	176	84	0.48	87	49	0.56	19	3	0.16	87	49	0.56
std::map::upper_bound	–	–	–	–	–	–	–	–	–	–	–	–
std::map::find	11	3	0.27	21	3	0.14	5	4	0.8	21	3	0.14
std::map::erase	4	4	1	4	4	1	4	4	1	4	4	1
std::map::at	4	3	0.75	4	3	0.75	1	0	0	4	3	0.75
std::vector::push_back	502	453	0.9	592	526	0.89	125	97	0.78	585	519	0.89
std::vector::resize	72	68	0.94	79	67	0.85	22	13	0.59	79	67	0.85
std::vector::clear	70	44	0.63	69	43	0.62	1	1	1	69	43	0.62
std::vector::reserve	3	0	0	10	0	0	2	0	0	10	0	0
std::vector::erase	11	5	0.45	11	5	0.45	6	1	0.17	11	5	0.45
std::vector::insert	5	4	0.80	6	4	0.67	5	4	0.80	6	4	0.67
std::vector::emplace_back	63	38	0.60	115	71	0.62	38	35	0.92	114	69	0.61
std::deque::push_back	43	39	0.91	48	39	0.81	6	5	0.83	48	39	0.81
std::deque::pop_back	3	2	0.67	3	0	0	–	–	–	3	0	0
std::deque::push_front	–	–	–	–	–	–	–	–	–	–	–	–
std::deque::pop_front	17	17	1	17	17	1	1	1	1	17	17	1
std::deque::operator[]	6	4	0.67	6	4	0.67	–	–	–	6	4	0.67
Overall	1161	837	0.72	1150	864	0.75	250	174	0.7	1142	855	0.75

Table 4: Results per method with unknown template parameters.

Function Name	Unknown template parameters											
	–O2			–O3			–Os			–Ofast		
	Found	Hit	Rate	Found	Hit	Rate	Found	Hit	Rate	Found	Hit	Rate
std::map::operator[]	470	205	0.44	466	171	0.37	53	8	0.15	466	171	0.37
std::map::lower_bound	580	332	0.57	567	322	0.57	73	16	0.22	567	322	0.57
std::map::upper_bound	43	14	0.33	36	16	0.44	21	7	0.33	36	16	0.44
std::map::find	190	14	0.07	253	25	0.10	75	7	0.09	253	25	0.10
std::map::erase	2	0	0	11	9	0.82	2	1	0.50	11	9	0.82
std::map::at	23	8	0.35	26	7	0.27	2	0	0	26	7	0.27
std::vector::push_back	1169	669	0.57	1380	833	0.60	215	114	0.53	1370	825	0.60
std::vector::resize	111	90	0.81	117	87	0.74	53	9	0.17	122	93	0.76
std::vector::clear	61	14	0.23	53	14	0.26	0	0	0	53	14	0.26
std::vector::reserve	33	1	0.03	54	1	0.02	35	1	0.03	53	1	0.02
std::vector::erase	40	0	0	27	1	0.04	16	4	0.25	27	1	0.04
std::vector::insert	8	0	0	18	2	0.11	8	1	0.12	18	2	0.11
std::vector::emplace_back	267	125	0.47	612	301	0.49	122	69	0.57	602	299	0.50
std::deque::push_back	58	41	0.71	76	31	0.41	22	14	0.64	76	31	0.41
std::deque::pop_back	–	–	–	–	–	–	–	–	–	–	–	–
std::deque::push_front	6	5	0.83	17	8	0.47	1	0	0	17	8	0.47
std::deque::pop_front	8	7	0.88	9	6	0.67	1	0	0	9	6	0.67
std::deque::operator[]	96	77	0.80	72	59	0.82	2	0	0	72	59	0.82
Overall	3165	1602	0.51	3794	1893	0.5	701	251	0.6	3778	1889	0.5

call it an *unknown template parameter*. To check whether an inline method is a known/unknown template parameter case, we extract the mangled name from the debugging information and look for the same mangled name in our query fingerprint database. In Table 3 and Table 4 we report the results for known and unknown template parameters respectively. We denote with $Found$ the number of methods inlined by the compiler, with Hit the number

of methods that BINO successfully recognizes and with *Rate* the hit rate, i.e., $Rate = \frac{Hit}{Found}$.

Most of the time, performance on known template parameters is higher: this is expected since we have fingerprints tailored for these specific cases. However, it is interesting that our method works even for unknown template parameters. This demonstrates how the fingerprints generated by BINO can generalize by capturing the most relevant features of the inline methods.

For basic built-in types, it is easy to see how they are likely to have similar CFGs and assembly instructions. More complex data types, and in particular user-defined classes, may behave differently. The methods of these classes could play an important role in the methods of the template classes we want to recognize. For instance, if we use a user-defined class as the template parameter of the `std::vector` class, the copy constructor of the user-defined class will be used by several methods of `std::vector` since any new object in the vector will come from a copy. In this case, the inline code of the vector methods includes the code of the user-defined class. Thus, our fingerprints may differ significantly from those with such a user-defined class as a template parameter.

On the other hand, the methods of the user-defined classes may not be inlined inside the method of the template class we want to recognize. This depends on the characteristics of the method: For methods that do not call the methods of the template parameters directly but instead rely on internal functions to call them, there is the chance that the assembly code of the methods of the template classes is not mixed up with the methods of the template parameters, especially if these internal functions are large enough (the compiler, as we mentioned, tends not to inline functions that are particularly large). In these cases, our method will work.

4.7. Fine Parameters Selection

As demonstrated in Section 4.4, appropriate values for the parameters S and M can be selected based on the specific class under consideration. For instance, when matching inline methods from the `std::map` class, we have achieved better results by selecting values for M and S as 5 and 0.84, respectively. However, it is noteworthy that individual methods may benefit from the use of more tailored values of M and S. To this end, for each method, we have determined the optimal values for M and S that maximize the F1-Score across all four optimization levels. The results obtained when employing the best parameter values for BINO are reported in Table 5. The

Table 5: Overall results with specific S and M per method.

Function Name		std::map::operator[]	std::map::lower_bound	std::map::upper_bound	std::map::find	std::map::erase	std::map::at	std::vector::push_back	std::vector::resize	std::vector::clear	std::vector::reserve	std::vector::erase	std::vector::insert	std::vector::emplace_back	std::deque::push_back	std::deque::pop_back	std::deque::push_front	std::deque::pop_front	std::deque::operator[]	Overall
S		0.75	0.86	0.88	0.84	0.9	0.75	0.75	0.89	0.82	0.75	0.82	0.75	0.77	0.82	0.94	0.75	0.81	0.75	
M		9	5	5	6	7	6	6	5	5	5	5	6	6	5	5	5	6	6	
–O2	**N**	1199	1431	65	371	7	27	3814	527	293	94	137	21	725	355	23	82	18	389	9578
	TPs	553	884	20	37	4	11	2963	366	123	6	15	8	354	279	12	71	13	300	6019
	FPs	83	125	17	2	28	0	526	348	38	0	37	0	277	192	0	16	2	263	1954
	Prec	0.87	0.88	0.54	0.95	0.12	1	0.85	0.51	0.76	1	0.29	1	0.56	0.59	1	0.82	0.87	0.53	0.75
	Rec	0.46	0.62	0.31	0.1	0.57	0.41	0.78	0.69	0.42	0.06	0.11	0.38	0.49	0.79	0.52	0.87	0.72	0.77	0.63
	F1	0.6	0.72	0.39	0.18	0.21	0.58	0.81	0.59	0.54	0.12	0.16	0.55	0.52	0.68	0.69	0.84	0.79	0.63	0.69
–O3	**N**	1054	1237	55	504	26	30	4311	540	287	137	129	40	1384	438	20	99	41	268	10600
	TPs	378	733	22	58	20	10	3300	340	115	7	19	10	754	272	5	86	8	211	6348
	FPs	0	121	27	71	107	0	570	392	23	0	29	0	305	188	0	15	8	135	1991
	Prec	1	0.86	0.45	0.45	0.16	1	0.85	0.46	0.83	1	0.4	1	0.71	0.59	1	0.85	0.5	0.61	0.76
	Rec	0.36	0.59	0.4	0.12	0.77	0.33	0.77	0.63	0.4	0.05	0.15	0.25	0.54	0.62	0.25	0.87	0.2	0.79	0.6
	F1	0.53	0.7	0.42	0.18	0.26	0.5	0.81	0.53	0.54	0.1	0.21	0.4	0.62	0.61	0.4	0.86	0.28	0.69	0.67
–Os	**N**	127	368	31	169	15	3	713	182	24	84	66	22	331	48	4	28	5	4	2224
	TPs	34	213	10	33	4	0	410	73	5	1	14	7	224	30	4	24	3	1	1090
	FPs	0	77	77	168	33	0	51	115	388	0	23	0	36	5	0	0	5	24	1002
	Prec	1	0.73	0.11	0.16	0.11	0	0.89	0.39	0.01	1	0.38	1	0.86	0.86	1	1	0.38	0.04	0.52
	Rec	0.27	0.58	0.32	0.2	0.27	0	0.58	0.4	0.21	0.01	0.21	0.32	0.68	0.62	1	0.86	0.6	0.25	0.49
	F1	0.42	0.65	0.17	0.18	0.15	0	0.7	0.39	0.02	0.02	0.27	0.48	0.76	0.72	1	0.92	0.46	0.07	0.51
–Ofast	**N**	1049	1231	55	504	26	30	4279	538	287	134	130	40	1373	438	20	99	41	268	10542
	TPs	376	731	22	58	20	10	3276	338	115	7	19	10	751	273	5	86	8	211	6316
	FPs	0	121	27	71	107	0	571	388	23	0	29	0	305	188	0	16	7	135	1988
	Prec	1	0.86	0.45	0.45	0.16	1	0.85	0.47	0.83	1	0.4	1	0.71	0.59	1	0.84	0.53	0.61	0.76
	Rec	0.36	0.59	0.4	0.12	0.77	0.33	0.77	0.63	0.4	0.05	0.15	0.25	0.55	0.62	0.25	0.87	0.2	0.79	0.6
	F1	0.53	0.7	0.42	0.18	0.26	0.5	0.81	0.53	0.54	0.1	0.21	0.4	0.62	0.61	0.4	0.86	0.29	0.69	0.67

results also highlights the performance improvements achieved by employing the optimal values of M and S for each method, as compared to using generic values for the entire class, or for the entire matching process.

4.8. Fingeprint Complexity Analysis

The performance of our approach varies depending on the complexity of the fingerprints of the method we are seeking. In Figure 7, we show how the precision depends on complexity, expressed in terms of the number of basic blocks that constitute the CFG within the fingerprint being analyzed. In most cases, the precision in recognition is positively correlated with the complexity: the higher the number of basic blocks, the higher the precision achieved. This is clear by looking at overall and `std::vector::push_back` precisions in Figure 7. This means that BINO is particularly reliable when considering complex methods. However, you can notice three significant sinks in the graph of overall precision. We can correlate those values with the

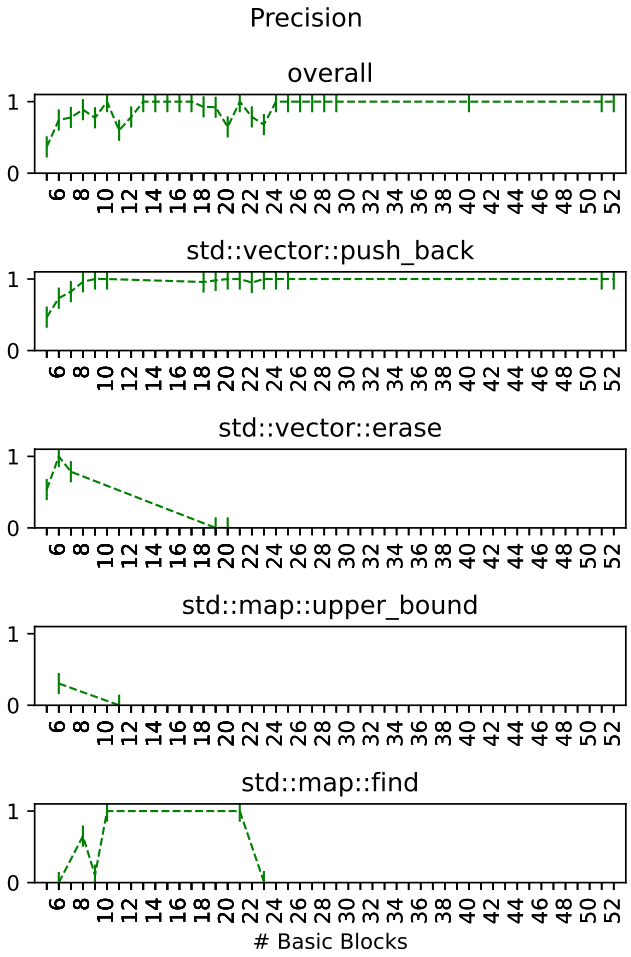


Figure 7: Precision per number of basic blocks

three methods `std::vector::erase`, `std::map::upper_bound`, and `std::map::find`. In fact, for those methods, the previous consideration seems not valid since the precision is zeroed as the number of basic blocks increases. This is caused by a high amount of false positives. We manually analyzed the false positives of these methods. The `std::vector::erase` and `std::map::find` methods are wrappers for internal functions (i.e., `std::vector::_M_erase` and `std::rb_tree::find|`). The matches classified as false positives are actually matching of the internal functions when not inlined. Similarly, the `std::map::upper_bound` method is a wrapper of `std::rb_tree::_M_upper_bound`, which is widely used in other template classes.

The false positives of the `std::map::upper_bound` method are matching the inline `std::rb_tree::_M_upper_bound` method of other classes. Although someone can argue that those should not be considered false positives, we choose the conservative approach and count them as false positives in all our experiments.

4.9. Impact of Syntactic and Semantic Features

In this experiment, we want to evaluate the impact of the semantic and syntactic features used by BINO. Indeed, we want to understand how each part of the fingerprint that we create contributes to the problem's solution. We have two features on top of the CFG: ❶ Function calls information, and ❷ Colors (i.e., mnemonic groups of assembly instructions). To understand their impact, we run three tests. One without any feature (the match is done considering only the CFG), one with only function calls, and one with only colors. It is important to note that the basic CFG matching is also present in the experiment for function calls and colors.

These tests are likely to generate many false positives that need to be confirmed through the debug information. Parsing DWARF debug information is particularly time-consuming. For this reason, we perform our test on a subset of the original dataset, made of 200 random projects that use the `std::vector` class. The choice of these projects is motivated by the high availability of projects that use such a class. For the same reason, we do not perform the tests with the `-Ofast` optimization level due to the fact that it has similar optimizations and performance compared to `-O3`. Instead, we consider the `-O2`, `-O3`, and `-Os` optimization levels for each test.

The F1-Scores of the three tests are shown in Table 6. For the test with only colors, we selected the S values that maximize the F1-Scores in the three optimization levels. Figure 8 shows the F1-Scores per optimization level with S ranging from 0.75 to 1. It is worth noting that the values of parameter S are higher compared to the value computed in Section 4.4. The reason for this is mainly attributed to the absence of function call information, leading to a higher number of false positives. Consequently, a higher similarity threshold is necessary to mitigate the effect of false positives.

You can immediately notice that the CFG alone cannot be used as an identification system and that the impact of colors is greater than the one of function calls information. This is mainly caused by the fact that the information of function calls is present only in the basic blocks with function calls, which are usually few across the basic blocks of the match. Moreover,

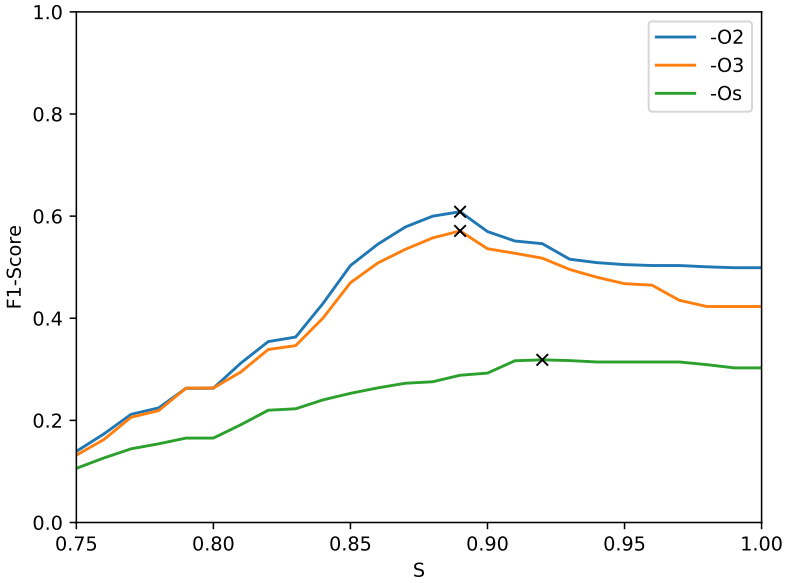


Figure 8: Similarity threshold (S) selection for only colors experiment. $\times$ refers to maximum points of the curves.

because of the multi-level inlining, some fingerprints, and their corresponding matches, do not have any function call. Instead, colors capture semantic information for each basic block, which is why colors have more impact with respect to function calls information. However, as demonstrated, they are both relevant for our analysis.

The impact of the features w.r.t. the number of basic blocks in a method is even more interesting. In Figure 9, we can observe the precision of BINO with different basic blocks amount. We notice that features have a significant impact when the amount of basic blocks is less than ten, whereas, with a high number of basic blocks, the CFG is enough to identify the functions univocally.

Table 6: BINO F1-Score per feature.

Optimization	**F1-Score**		
	CFG	**Function calls**	**Colors**
-O2	<0.01	0.12	0.61
-O3	<0.01	0.1	0.57
-Os	<0.01	0.04	0.32

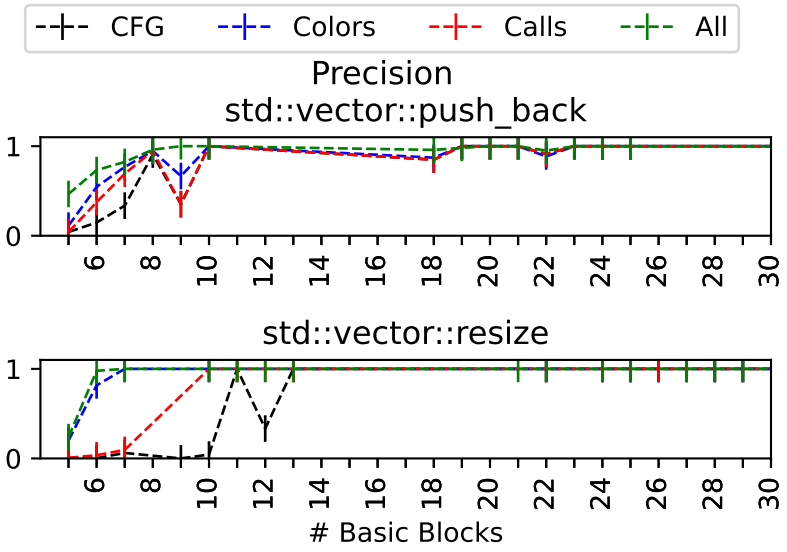
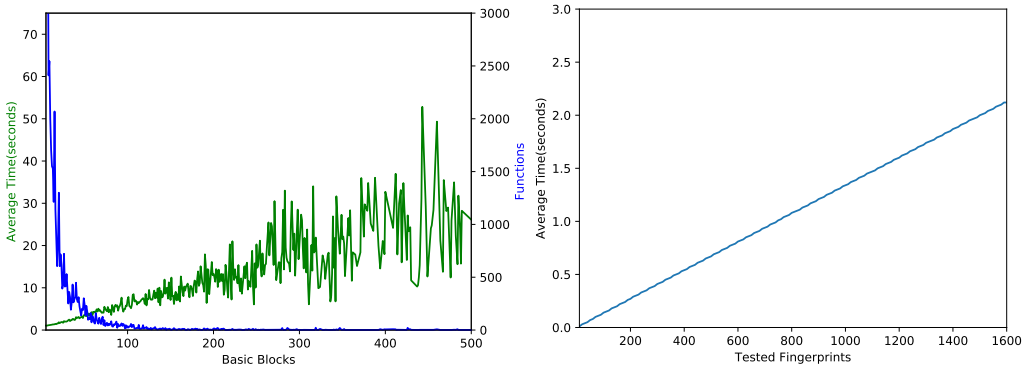


Figure 9: Impact of features w.r.t. the number of basic blocks

4.10. Scalability

Scalability is a significant challenge in our approach, as memory and computational requirements increase with the number of query fingerprints. Figure 10a illustrates the relationship between matching time and function size, measured in the number of basic blocks. Notably, most functions are small (blue line in Figure 10a), resulting in an average fingerprint matching time of 1.33 ms. Additionally, Figure 10b demonstrates the relationship between matching time and the number of fingerprints in our database. As shown, the average matching time increases linearly with the number of employed fingerprints.

We remind the reader that we need to generate a set of fingerprints for each target combination of template class and datatype(s) in a template parameter. For template classes with multiple template parameters (e.g., `std::map`), this results in a combinatorial explosion. The impact in our experiments has been limited because we considered only the most intuitive and common types used by these classes, but making our approach scale to a complete coverage will be challenging.



(a) Average matching time and functions amount w.r.t. the number of basic blocks.

(b) Average matching time w.r.t. the number of fingerprints in the fingerprints database.

Figure 10: Scalability

5. Limitations and future works

Our approach has lower performance with methods composed of few ($M < 5$) basic blocks (e.g., getter or setter methods in objects). When such methods are inlined, they translate into a few assembly instructions that generate a high level of irrelevant matches, i.e., false positives. This phenomenon is clear from Figure 7 and Figure 9. Although this limitation impacts the number of methods that we are to handle, it affects only small methods that are easier to reverse. Nevertheless, our approach performs well on medium and big methods that are cumbersome to reverse engineers.

Moreover, when inlining, a compiler, which runs aggressive optimization passes and infers the status of objects and the value of variables, can drastically simplify the execution flow of the inlined assembly code (e.g., by shorting out irrelevant branches). This can modify the CFG enough to thwart recognition by the subgraph isomorphism algorithm.

While not exactly a limitation, it is important to note that the detection of the boundaries of inlined methods is necessarily imprecise since it has the granularity level of basic blocks. Since, when inlining, the callee function body is melded within the caller and then further optimized by the compiler, the "boundary" basic blocks are a mix of instructions from both the callee and the caller. Further work and research would be needed to further improve the granularity of the boundary detection of the inlined method.

A potential extension of the work would be to automate the extraction of further information relevant to reverse engineering once a match is found.

For instance, BINO could automatically derive the memory address of the object, and which register(s) hold a reference to it. This information could be further used to track inline methods (such as the small ones described before) that we have otherwise missed by detecting memory access to locations at a certain offset from the detected base memory address. We could also track its creation and destruction, as well as inter-procedural sources and sinks involving it.

Moreover, we foresee the application of BINO to a multi architecture dataset. This can be achieved by defining mnemonic groups (colors) for each new architecture. It would be interesting to verify if fingerprints are transferable across different architectures.

6. Related works

Reverse engineering is a complex process. The multiple steps that lead from a sequence of bytes to the understanding of a program's underlying semantics and logic are complex, sometimes based on heuristic or human intuition, and rely on a series of assumptions. Relaxing such assumptions, improving automation, and helping analysts is a challenging and complex research field.

For instance, a basic assumption in reversing is that the architecture the code is designed for is known. While this is often true (e.g., because the binary file's header offers clues about it), this is not necessarily a given when analyzing embedded firmware. For this reason, multiple works [18, 19] showed that it is possible to correctly guess the architecture from the byte sequence of the binary.

The second step in the reverse engineering process is disassembling the byte sequence into instructions. Disassembling has its challenges, but there are several works [20, 21, 2] that show how this is usually doable.

The third step is the identification of the boundaries of a function. This is a complex challenge because the concept of "function" is an abstraction that is lost during compilation, as boundaries are not necessary for binaries to work, and this information is usually sacrificed to performance. Recent works [22] showed how it is possible to identify such boundaries with a reasonable level of precision. There is, however, a prominent case where this approach systematically fails: inline functions, which is the focus of our work.

A commonly adopted approach to assist and speed up reverse engineering is identifying known functions. Several research works [8, 23, 10, 24] and tools

(such as IDA FLIRT [25]) try to detect the usage of library functions. You should read [26] for a complete overview of binary similarity approaches, and [27] for an overview of Machine Learning approaches.

One of the most promising approach for function clone identification is *kam1n0* [28]. *kam1n0* has some similarity w.r.t. BINO; both approaches relies on subgraph isomorphism algorithms to identify assembly clones. BINO relies on *VF2* [15, 16] to identify subgraph isomorphic CFGs, while *kam1n0* leaverages on Local Sensitive Hashing (LSH) and MapReduce to identify isomorphic subgraphs with similar semantic and syntactic instructions. The LSH algorithm clusters the basic blocks of the binary, while MapReduce constructs isomorphic subgraphs by iteratively aggregating basic blocks that are in the same cluster. Despite the similarities in approach, the two methods address different problems. *Kam1n0* identifies subgraph isomorphic CFGs between two functions, while BINO identifies all the inline method calls in binary functions. Hence, *kam1n0* fails by design when one or more method calls are inlined within a single function. Moreover, the LSH algorithm fails when clustering the initial and final basic blocks of inline methods since they contains extra instructions of the caller.

Some works [29, 30] can identify special-purpose functions such as Custom Memory Allocators. However, all these techniques critically depend on correctly identifying function boundaries and, thus, fail when dealing with inlined functions. Our approach is unique not just in not needing function boundaries but also in being able to identify library functions blended with other code.

To achieve such a detection system, we create a fingerprinting system to capture structural and semantic information of functions (as seen in Section 3). A similar concept of fingerprinting in binaries was developed in [11], where the resilience of such fingerprints was shown to be robust against polymorphism in worms. In [31] the same concept was paired with user-defined behaviors of malware (i.e., with dynamically displayed functionalities). This led to the use of fingerprinting on malicious binaries to recognize malware development efforts in [32], and to the automatic analysis and extraction of both dynamic behaviors of interest and related binary code fingerprints in [12].

7. Conclusions

Recognition of inline functions is a very complex and challenging topic, but it would be extremely helpful for reverse engineers, particularly if applicable to methods of well-known template classes.

With this work, we proposed an approach to this problem based on the concept of a fingerprint. Around this concept, we developed a framework to generate and compare our fingerprints to recognize inline library functions. In addition, we provided a methodology to automate the generation of fingerprints starting from the parsing of the library source code. During the development of BINO, we studied in-depth the process of inlining, and we analyzed the common aspects and features to be used in our fingerprint model.

Our results on a dataset of C++ GitHub projects, compiled with different optimization levels, show that BINO can match methods from classes such as `std::map` and `std::vector` with an F1-Score of 65%, with `-O2` optimization level. This value is similar for the `-O3/-Ofast` optimization levels and it is lower for the `-Os` optimization level.

Our approach, although promising, has a few known limitations. Some of them depend purely on the implementation; others are more structural. Detecting small functions is challenging. Scalability can become an issue. The current granularity is at the basic block level and cannot determine the specific point within a basic block where the inline function starts or ends.

This work could be extended to automatically gather more information about the inline function (some of which may be used to overcome some of the aforementioned limitations). We believe that full implementation of this approach would be exceptionally helpful to reverse engineers.

References

[1] Dwarf debugging information format version 5, published standard (2017) [cited August 16th, 2021].
URL `http://dwarfstd.org/doc/DWARF5.pdf`

[2] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, G. Vigna, SOK: (state of) the art of war: Offensive techniques in binary analysis, in: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016, IEEE Computer Society, 2016, pp. 138–157.

doi:10.1109/SP.2016.17.
URL https://doi.org/10.1109/SP.2016.17

[3] A. D. Federico, M. Payer, G. Agosta, rev.ng: a unified binary analysis framework to recover cfgs and function boundaries, in: P. Wu, S. Hack (Eds.), Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017, ACM, 2017, pp. 131–141.
URL http://dl.acm.org/citation.cfm?id=3033028

[4] H. Koo, S. Park, T. Kim, Revisiting function identification with machine learning, in: Machine Learning for Program Aanalysis (MLPA) Workshop, 2021.

[5] K. Kennedy, J. R. Allen, Optimizing compilers for modern architectures: a dependence-based approach, Morgan Kaufmann Publishers Inc., 2001.

[6] Y. Ben-Asher, O. Boehm, D. Citron, G. Haber, M. Klausner, R. Levin, Y. Shajrawi, Aggressive function inlining: Preventing loop blockings in the instruction cache, in: P. Stenström, M. Dubois, M. Katevenis, R. Gupta, T. Ungerer (Eds.), High Performance Embedded Architectures and Compilers, Third International Conference, HiPEAC 2008, Göteborg, Sweden, January 27-29, 2008, Proceedings, Vol. 4917 of Lecture Notes in Computer Science, Springer, 2008, pp. 384–397. doi:10.1007/978-3-540-77560-7_26.
URL https://doi.org/10.1007/978-3-540-77560-7_26

[7] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, H. B. K. Tan, Bingo: cross-architecture cross-os binary search, in: T. Zimmermann, J. Cleland-Huang, Z. Su (Eds.), Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016, ACM, 2016, pp. 678–689. doi:10.1145/2950290.2950350.
URL https://doi.org/10.1145/2950290.2950350

[8] S. H. H. Ding, B. C. M. Fung, P. Charland, Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization, in: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019, IEEE,

2019, pp. 472–489. doi:10.1109/SP.2019.00003.
URL https://doi.org/10.1109/SP.2019.00003

[9] Y. Duan, X. Li, J. Wang, H. Yin, Deepbindiff: Learning program-wide code representations for binary diffing, in: 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020, The Internet Society, 2020.

[10] L. Nouh, A. Rahimian, D. Mouheb, M. Debbabi, A. Hanna, Binsign: Fingerprinting binary functions to support automated analysis of code executables, in: S. D. C. di Vimercati, F. Martinelli (Eds.), ICT Systems Security and Privacy Protection - 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29-31, 2017, Proceedings, Vol. 502 of IFIP Advances in Information and Communication Technology, Springer, 2017, pp. 341–355. doi:10.1007/978-3-319-58469-0_23.
URL https://doi.org/10.1007/978-3-319-58469-0_23

[11] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, G. Vigna, Polymorphic worm detection using structural information of executables, in: RAID, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 207–226.

[12] M. Polino, A. Scorti, F. Maggi, S. Zanero, Jackdaw: Towards automatic reverse engineering of large datasets of binaries, in: M. Almgren, V. Gulisano, F. Maggi (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings, Vol. 9148 of Lecture Notes in Computer Science, Springer, 2015, pp. 121–143. doi:10.1007/978-3-319-20550-2_7.
URL https://doi.org/10.1007/978-3-319-20550-2_7

[13] M. Grohe, P. Schweitzer, The graph isomorphism problem, Commun. ACM 63 (11) (2020) 128–134. doi:10.1145/3372123.
URL https://doi.org/10.1145/3372123

[14] S. Fortin, The graph isomorphism problem, Tech. Rep. TR96-20, University of Alberta (1996).

[15] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, A (sub) graph isomorphism algorithm for matching large graphs, IEEE transactions on pattern analysis and machine intelligence 26 (10) (2004) 1367–1372.

doi:10.1109/TPAMI.2004.75.
URL https://doi.org/10.1109/TPAMI.2004.75

[16] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, An improved algorithm for matching large graphs, in: 3rd IAPR-TC15 workshop on graph-based representations in pattern recognition, 2001, pp. 149–159.

[17] J. Lee, W. Han, R. Kasperovics, J. Lee, An in-depth comparison of subgraph isomorphism algorithms in graph databases, Proc. VLDB Endow. 6 (2) (2012) 133–144. doi:10.14778/2535568.2448946.
URL http://www.vldb.org/pvldb/vol6/p133-han.pdf

[18] P. D. Nicolao, M. Pogliani, M. Polino, M. Carminati, D. Quarta, S. Zanero, ELISA: eliciting ISA of raw binaries for fine-grained code and data separation, in: C. Giuffrida, S. Bardin, G. Blanc (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings, Vol. 10885 of Lecture Notes in Computer Science, Springer, 2018, pp. 351–371. doi:10.1007/978-3-319-93411-2_16.
URL https://doi.org/10.1007/978-3-319-93411-2_16

[19] S. Kairajärvi, A. Costin, T. Hämäläinen, Isadetect: Usable automated detection of cpu architecture and endianness for executable binary files and object code, in: Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy, Association for Computing Machinery, New York, NY, USA, 2020, p. 376–380.
URL https://doi.org/10.1145/3374664.3375742

[20] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, G. Vigna, Ramblr: Making reassembly great again, in: 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017, The Internet Society, 2017.
URL https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/ramblr-making-reassembly-great-again/

[21] D. Brumley, I. Jager, T. Avgerinos, E. J. Schwartz, BAP: A binary analysis platform, in: G. Gopalakrishnan, S. Qadeer (Eds.), Computer Aided Verification - 23rd International Conference, CAV 2011,

Snowbird, UT, USA, July 14-20, 2011. Proceedings, Vol. 6806 of Lecture Notes in Computer Science, Springer, 2011, pp. 463–469. doi:10.1007/978-3-642-22110-1_37.
URL https://doi.org/10.1007/978-3-642-22110-1_37

[22] T. Bao, J. Burket, M. Woo, R. Turner, D. Brumley, BYTEWEIGHT: learning to recognize functions in binary code, in: K. Fu, J. Jung (Eds.), Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014, USENIX Association, 2014, pp. 845–860.
URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bao

[23] P. Shirani, L. Wang, M. Debbabi, Binshape: Scalable and robust binary library function identification using function shape, in: M. Polychronakis, M. Meier (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings, Vol. 10327 of Lecture Notes in Computer Science, Springer, 2017, pp. 301–324. doi:10.1007/978-3-319-60876-1_14.
URL https://doi.org/10.1007/978-3-319-60876-1_14

[24] S. H. H. Ding, B. C. M. Fung, P. Charland, Kam1n0: Mapreduce-based assembly clone search for reverse engineering, in: B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, R. Rastogi (Eds.), Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016, ACM, 2016, pp. 461–470. doi:10.1145/2939672.2939719.
URL https://doi.org/10.1145/2939672.2939719

[25] IDA F.L.I.R.T. technology: In-depth.
URL https://hex-rays.com/products/ida/tech/flirt/in_depth/

[26] I. U. Haq, J. Caballero, A survey of binary code similarity, ACM Comput. Surv. 54 (3) (apr 2021). doi:10.1145/3446371.
URL https://doi.org/10.1145/3446371

[27] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, D. Balzarotti, How machine learning is solving the binary function similarity problem, in: 31st USENIX Security Symposium (USENIX Security 22), USENIX Association, Boston, MA, 2022.

URL https://www.usenix.org/conference/usenixsecurity22/presentation/marcelli

[28] S. H. Ding, B. C. Fung, P. Charland, Kam1n0: Mapreduce-based assembly clone search for reverse engineering, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 461–470. doi:10.1145/2939672.2939719.
URL https://doi.org/10.1145/2939672.2939719

[29] X. Chen, A. Slowinska, H. Bos, Who allocated my memory? detecting custom memory allocators in C binaries, in: R. Lämmel, R. Oliveto, R. Robbes (Eds.), 20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013, IEEE Computer Society, 2013, pp. 22–31. doi:10.1109/WCRE.2013.6671277.
URL https://doi.org/10.1109/WCRE.2013.6671277

[30] X. Chen, A. Slowinska, H. Bos, On the detection of custom memory allocators in C binaries, Empir. Softw. Eng. 21 (3) (2016) 753–777. doi:10.1007/s10664-015-9362-z.
URL https://doi.org/10.1007/s10664-015-9362-z

[31] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, S. Zanero, Identifying dormant functionality in malware programs, in: SP, IEEE Computer Society, Washington, DC, USA, 2010, pp. 61–76.

[32] M. Lindorfer, A. D. Federico, F. Maggi, P. M. Comparetti, S. Zanero, Lines of malicious code: Insights into the malicious software industry, in: ACSAC, ACM, New York, NY, USA, 2012, pp. 349–358.

Appendix A. Fingerprint Features

This appendix provides additional information on the mnemonic groups and the function call information used by BINO.

The mnemonic groups used are listed in Table A.7. We purposely omitted data transfer instructions for two reasons: first, they are present in almost any given basic block; second, since the compiler can rearrange `mov` operations easily even to a different basic block, they are a very brittle indicator.

We distinguish four different types of calls, described in Table A.8. Moreover, for standard and library calls, we also annote the name and the path

Table A.7: Mnemonic groups of assembly instructions used in the `color` bit vector of fingerprints.

Mnemonic Group	**Description**
Arithmetic	Instructions like `add`, `sub`, `mul`, and so on.
Branch	Instructions like `jnz`, `jne`, `jae`, and so on.
Call	Instructions like `jnz`, `jne`, `jae`, and so on.
Cond Move	Instructions like `cmova`, `cmovb`, `cmovc`, and so on.
Flags	Instructions like `btc`, `clc`, `cld`, and so on.
Float	Instructions like `fadd`, `fsub`, `fmul`, and so on.
Halt	The `hlt` instruction.
Interleaving	Instructions that apply masks to the register or reorder the bits in the registers, such as `shufps` and `pshufd`.
Jump	The `jmp` instruction.
Lea	The `lea` instruction.
Logic	Instructions like `xor`, `and`, `or`, and so on.
Misc	Instructions like `in`, `out`, `leave`, and so on.
Sign	Instructions like `cbw`, `cdq`, `cwd`, and so on. Mainly used for sign extensions.
Stack	Instructions like `pop` and `push`.
Syscall	The `syscall` instruction.
String	Instructions like `cmpsd`, `repmovsb`, `stosq`, and so on. Mainly used for strings management.
Test	Instructions like `cmp` and `test`.

of the called functions. As a recall, the path of a function or method in C++ specifies its location within a namespace, a class, or both. For instance, the `std::vector::push_back` method has `push_back` as name and `std::vector` as path. Instead, the `std::copy` utility function has `copy` as name and `std` as path. Finally, C library functions, such as `printf` and `scanf`, do not have a path.

Appendix B. CFG Variations

This appendix describes how the BINO recognizes inline methods even their corresponding query fingerprints have not the same structure. Depending on where the call to the method is performed and due to optimizations applied by the compiler, the generated CFG can change significantly. We automatically handle the three common cases shown in Figure B.11. We denote with I the initial basic block and with F the final basic block. The CFG on the top left is our query fingerprint. The first case of the behavior described

Table A.8: Jump types used for function calls feature.

Jump Type	**Description**
Standard	A call to another function of the binary.
Library	A call to a `plt` stub. Thus, a call that will execute a function located in a shared library.
Indirect	A call to an address which is stored in a register at runtime. For instance `call rax`.
Syscall	A call to request a service from the kernel. In `x86` it is either `syscall` or `int 0x80`.

above happens when a function call is placed inside a loop. In such a case, the last basic block of the method, once inlined, contains a few additional assembly instructions: an increment of a value in a register, a comparison with a constant or another register, but most importantly for our matching process, a conditional jump to the first basic block of the inlined method. This adds an edge between the last and the first basic block, as shown in variation 2 in Figure B.11. To handle this, we add the edge from the last basic block to the first basic block before performing the subgraph isomorphism algorithm. A second common case happens when the next to last basic block(s) performs an unconditional jump to the final basic block. In this case, the compiler may move (depending on the size of F) the assembly instructions belonging to the final block into the previous one(s) to get rid of the jump, as shown in variation 3 in Figure B.11. To handle this, we remove the last basic block from the query CFG when the next-to-last basic block has an unconditional jump before performing the subgraph isomorphism algorithm. The two cases can also be combined (i.e., the next to last basic block may be performing a conditional jump to the first, as shown in variation 4 in Figure B.11). We handle this by combining the two procedures outlined above: we remove the final basic block and add an edge from the next-to-last basic block (which now is the final one) to the initial basic block.

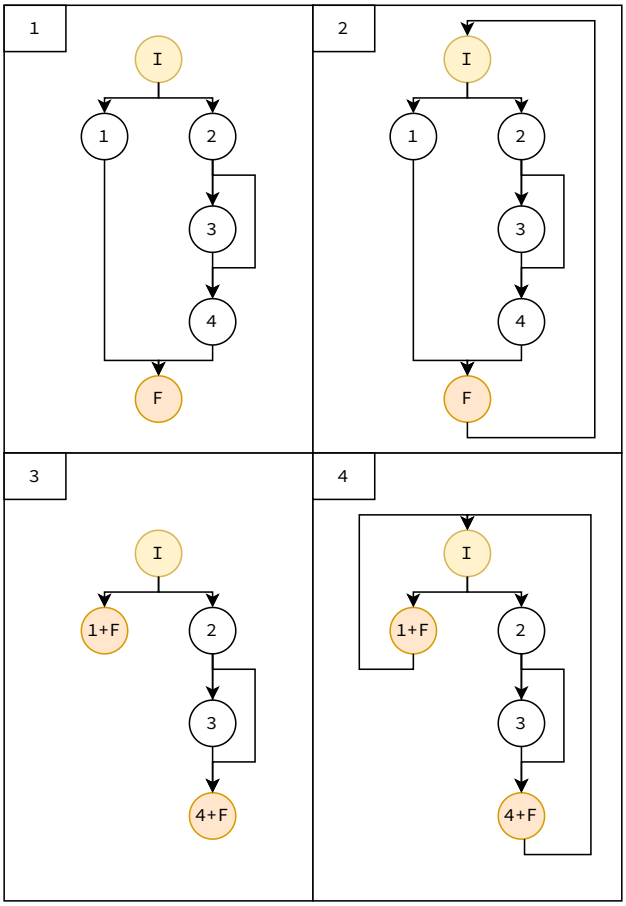


Figure B.11: A query CFG (number 1, top left), and three common variations we specifically handle.