# Enhancing Control Flow Graph Based Binary Function Identification

Clemens Jonischkeit
Technical University of Munich
Garching, Germany
jonischk@sec.in.tum.de

Julian Kirsch
Technical University of Munich
Garching, Germany
kirschju@sec.in.tum.de

## ABSTRACT

Recognition of binary functions in compiled code is a major stepping stone towards any advanced binary analysis technique. Nucleus [**?** ] is a novel algorithm based on the idea of using the Interprocedural Control Flow Graph (ICFG) to detect function boundaries. Building upon this technology we propose a new approach to address the related problem of *identifying* previously-seen known functions within a binary. Our idea is based on comparing the Control Flow Graphs (CFGs) of unknown functions from a binary to known functions from a previously generated database. Compared to traditional approaches, our method is *aware of the underlying graph matching problem being performed on CFGs of binary code*: First, it utilizes instruction level knowledge about basic blocks as additional constraints for graph isomorphism. Second, optimizations and transformations introduced by different compilers affecting the shape of the CFG are taken into account.

Our approach aims to avoid false positives (wrongly assigning a known function symbol to an unknown function) at all cost: The evaluation shows that this method is very effective in reducing false positive matches (below one percent in most cases) and doubles recall rates compared to the traditional graph matching based approach when matching one version of *nginx* compiled with different optimization levels.

## CCS CONCEPTS

• **Security and privacy → Software reverse engineering**;

## KEYWORDS

Binary Function Identification, Control Flow Graph, Graph Matching

**ACM Reference Format:**
Clemens Jonischkeit and Julian Kirsch. 2017. Enhancing Control Flow Graph Based Binary Function Identification. In *ROOTS: Reversing and Offensive-oriented Trends Symposium, November 16–17, 2017, Vienna, Austria.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3150376.3150384


*ROOTS, November 16–17, 2017, Vienna, Austria*

ACM ISBN 978-1-4503-5321-2/17/11…$15.00
https://doi.org/10.1145/3150376.3150384

## 1 INTRODUCTION

Function recognition is a fundamental building block in reverse engineering. It is a necessity for many advanced analysis techniques, like binary instrumentation or the detection of software vulnerabilities. The goal of function recognition is to group instructions to resemble source level functions. This can easily be done in situations where symbol data for a binary is available but is challenging when no such information is present.

While identifying the exact position of functions is a crucial step during reverse engineering a binary, reverse engineers find themselves to spend significant amounts of time to analyze the same function in any unknown binary over and over again. One purpose of function identification is consequently to save analysts from wasting their time determining that functions in an unknown binary are equivalent to (potentially known) functions previously encountered earlier in a different binary. With function information available, a reverse engineer can more efficiently decide which parts of an unknown binary are of interest to look at, as the functionality of library functions statically linked into a binary is already known. For example, analysts looking for software vulnerabilities could focus on closely analyzing code that calls potentially dangerous or error prone library functions like the `strcpy` function from `glibc`.

For dynamically linked executables this task is trivial as the external function name has to be known to the dynamic loader: When resolving an external symbol, the dynamic loader looks up the address of the symbol in the library and patches the executable such that calls are dispatched correctly. On the other hand, for statically linked functions no information has to be preserved as the referenced library code is included into the binary making addresses and jump distances known at compile time. The linker can then strip the names of the library symbols without affecting the correct functionality of the binary, resulting in an additional analysis step for a reverse engineer.

Therefore, with no information about the type of functions being present in a particular binary, function recognition methods need to be developed: Function boundaries have to be accurately detected before matching candidates against a database containing information about well known functions—in case of a successful match the unknown function ideally has the same semantics as the candidate from the database. We note that matching functions based on their semantics alone is theoretically *impossible in generality* as it would entail solving the halting problem. For this reason, function identification is always done based on heuristics and any matching approach requiring finite time is generally not sound, incomplete or both.

In this work, we propose a graph matching based approach to recognize known functions in unknown binaries. As opposed to

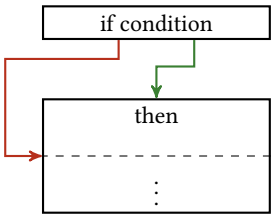


**Figure 1: Two basic blocks forming an if–then statement**

other graph matching based methods, our central contribution is to make the matching algorithm aware of the fact that it is operating on graphs consisting of basic blocks containing assembly code: First, we keep instruction information with the basic blocks stored in the database, and second control flow graphs are compared in a *O–O2 (n.)* form that takes into account different transformations performed by different compilers operating in different optimization levels.

## 2 BACKGROUND

In this section we will briefly highlight concepts we base our work on.

### 2.1 Basic Blocks

We will use the notion of multiple entry single exit basic blocks throughout this work: A basic block is a (ordered) list of consecutive machine instructions. Within these basic blocks, all instructions spanning from an entry point to the last instruction of the block will be executed (i.e. the block will not be left early) resulting in a single exit point, right after the last instruction. Unlike exit points it *is* possible for a basic block to have multiple different entry points. This implies that there can be edges pointing into the middle of a basic block, such that instructions at the beginning are skipped. With this notation an `if-then` construct can be translated into two basic blocks, with the second one having two entry points. This can be seen in figure 1. All basic blocks belonging to one function together with the edges introduced by control flow changing instructions form the static CFG of this function.

### 2.2 Graph Theory and Terminology

A CFG is a directed graph $G$ consisting of a set of vertices $V$ and a set of edges $E \subseteq V \times V$. An edge $e = (v_i, v_j) \in E$ represents a possible transition from vertex $v_i$ to vertex $v_j$, but not vice versa. The vertex $v_i$ is said to be the direct *ancestor* of $v_j$, whereas $v_j$ is called the *descendant* of $v_i$.

All vertices $v_i \in G$ connected by a path, regardless of the direction of the edges, form a weakly connected component of $G$. An undirected path connects all vertices within this weakly connected component. Each weakly connected component is a subgraph of $G$ and $G$ can have multiple weakly connected components.

We will call vertices $v_i \in G$ without a descendant ($\forall v_j \in V.(v_i, v_j) \notin E$) *leaf vertices*, to account for the fact that control flow moves to a different function after the execution of such a vertex.

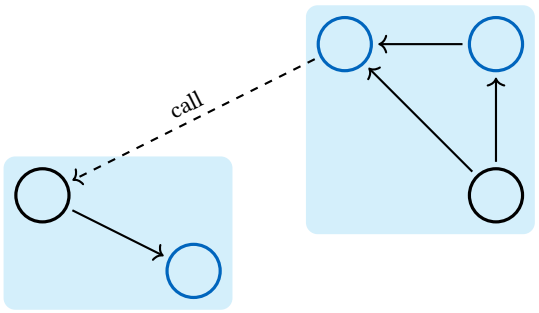


**Figure 2: Analysis of an ICFG, splitting it into components and identifying the entry points**

Two Graphs $G$ and $H$ are called isomorph, iff there exists a bijection $f : V(G) \to V(H)$ between the vertices $V(G)$ of $G$ and $V(H)$ of $H$, such that $(a, b) \in E(G) \Leftrightarrow (f(a), f(b)) \in E(H)$.

Furthermore, throughout this work, we will use the term *vertex* and *node* interchangeably to refer to a basic block within a function.

## 3 GRAPH-BASED FUNCTION IDENTIFICATION

Like Nucleus [**?** ] utilizes the ICFG to reliably detect function boundaries, the CFGs can be used to identify functions. The idea is that with an increasing number of basic blocks, the possible number of CFGs grows exponentially making large functions identifiable by their unique CFG. Additional measures, like the size, call targets, and referenced strings, can be used to match more strictly than just graph isomorphism. Our approach is based on Nucleus to *recognize* function boundaries, which we briefly explain before we divert out our algorithm to *identify* the recognized functions.

### 3.1 Function Recognition

Nucleus is a newly proposed algorithm for function recognition. It is entirely different from previous approaches as it does not rely on any (implicit or explicit) function signature databases. It attempts to overcome the difficulties and disadvantages of approaches based on byte patterns. Instead of searching for predetermined byte strings, it uses the observation that (in compiler generated code) interprocedural calls differ from intra procedural jumps.

Nucleus' algorithm for graph based function recognition can be broken down into four steps. First the ICFG is generated. A basic block in the executable is represented by a vertex in the graph and an edge represents a control flow transfer, examples of these transfers are jumps, calls, and return instructions. This graph is generated by disassembling and subsequently grouping instructions into basic blocks and then analyzing the control flow. It is next preprocessed to improve the analysis.

Temporarily hiding all call edges facilitates a weakly connected components analysis. Figure 2 shows an example of these components. The goal of this analysis is to find groups of basic blocks, such that each group of basic blocks contains all yet only basic blocks belonging to a single function. Hiding the call edges removes all inter procedural edges from the graph leaving only the intra procedural edges, with some exceptions, in the graph. Without edges between

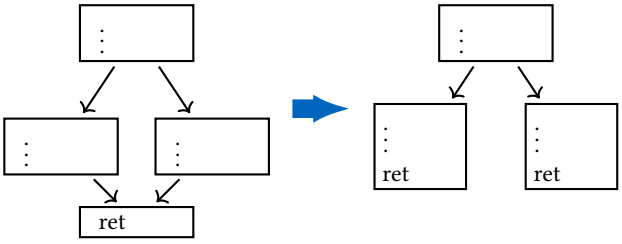


**Figure 3: Leaf Node Inlining**

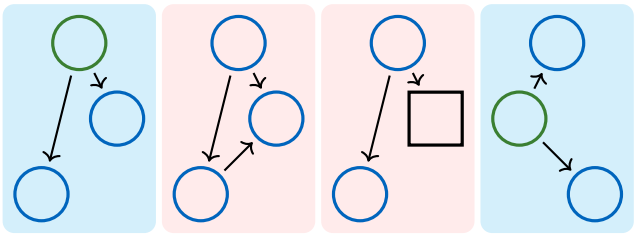

**Figure 4: Graphs compared against the first one, color and shape of a node indicate different meta data**

functions the ICFG can be split into weakly connected components, each representing one function.

Once the components have been identified their entry blocks have to be determined. Scanning over the instructions of all basic blocks reveals direct calls using the `call` instruction. Basic blocks targeted by these calls are expanded to full functions by following control flow edges until a weakly connected component is formed. After this step all components building a directly called function are known.

In a last step an attempt is made to find unreachable functions (e.g. functions that are never called) and indirect calls by expanding all basic blocks that do not belong to any function into one like in the step before. Finally, the entry point of the function is determined using different heuristics.

## 3.2 Function Identification

After recognizing functions within the target binary, our algorithm to identify known functions in unknown binaries is performed. It consists of four steps (Reconnaissance, Normalizing, Graph Matching, and Proximity Analysis) which we describe in the following.

*3.2.1 Reconnaissance.* In a first step, we use Nucleus (a) to solve the base problem of finding functions in a particular binary and (b) to obtain the boundaries of the basic blocks of each function. We slightly modified Nucleus to extract additional information, such as the number of instructions in each basic block. However, the core functionality and the algorithm remain untouched. With this information at hand, a function can be represented as a set of basic blocks associated with additional meta information. In context of our work, this additional meta information currently comprises two features: First, a list of called targets (if any), and second the number of instructions. This list of features can be extended to allow for a more exact matching or even to loosen up the strict matching. Based on the information recovered by Nucleus and the intra procedural edges a CFG is generated for each function.

*3.2.2 Normalizing.* Once the CFG and the features have been extracted from the binary by Nucleus the resulting data is transformed during a normalization step to allow for a more versatile matching. Concretely, the function graphs are transformed in two ways:

First, in an attempt to cope with different compiler and optimization levels leaf vertices in the CFG are duplicated and then merged into the respective preceding basic block(s). The reason is that *gcc* with optimization level one tends to create a common sink when control flow should return from a function, while *clang* using optimization level two emits multiple nodes from which the function might return. To increase the probability of a successful match leaf nodes are duplicated and merged into blocks which only have the (returning) leaf vertex as descendant. The resulting basic block does not exist in this way in the binary but it is semantically equivalent to the version consisting of the two merged blocks. In this step meta data from both blocks is coalesced as well to keep the block size and other measures consistent. This procedure is illustrated in Figure 3.

The second transformation being performed concerns merging of some nodes. In many cases the generated function CFGs contain basic blocks that have only one ancestor. If such a block does not have any other incoming edges, both basic blocks can be merged. In particular, such mergeable blocks occur for two reasons:

First, Nucleus treats function calls as exits from a particular basic block, resulting in blocks immediately ending after `call` instructions. Furthermore, if two calls to side effect free functions do not depend on each other then their order in the binary does not matter for the correctness of the function result. Consequently, both calls could be emitted at compile time in any order. Merging these two nodes allows the graph checker to account for *compiler introduced function call reordering*.

Second, distinct basic blocks can be mergeable because of `jmp` instructions: A basic block targeted by a direct, unconditional jump can be merged into its ancestor if the edge represented by the jump is the only connection of the ancestor and descendant. Since the next instruction after these kind of jumps is always the same, both blocks can be seen as one large block since the sequence of instructions executed is always the same regardless of the presence of the control flow change. It is important to note that merging only occurs if the block targeted by the jump has no other incoming edges, like the fall through edge of conditional jumps and jumps in general.

The purpose of the normalizing step is to make the matching phase more resistant while matching across different compilers, compiler versions, code generation strategies, optimization levels, or binary versions. However, normalization introduces ambiguities that can decrease the rate of successful matches. Therefore, in case the exact version of a binary is known, the normalization step can be omitted.

*3.2.3 Graph Matching.* Once the CFGs of all functions have been determined and (optionally) normalized, the actual matching of functions is performed. For two CFGs to be considered similar, they not only have to be isomorph, but also the meta data of the

**Algorithm 1:** Finding an isomorph mapping between graphs

```
function match (n1, n2, mapping);
  Input  : One Node from each graph n1, n2, and a mapping
           between the nodes of the graphs
  Output : A list of mappings between nodes of the graphs
nMap := mapping;
if n1 ∉ mapping.keys() then
    if !check(n1, n2) || mapping (suc1 n1) ⊈ suc2 n2 then
        return [];
    end
    nMap := mapping ∪{(n1, n2)};
end
sN1 := [a | a ← suc1 n1, a ∉ nMap.keys()];
sN2 := [a | a ← suc2 n2, a ∉ nMap.elems()];
if sN1 = [] then
    if nMap (suc1 n1) = suc2 n2 then
        return [nMap];
    else
        return [];
    end
end
nxt := concat([match(sN1.first(), tar, nMap) | tar ← sN2]);
return concat([match(n1, n2, resMap) | resMap ← nxt]);
```

basic blocks has to match. For example, in Figure 4 the first, third, and fourth graphs are isomorph, but the third one has a rectangle node representing different meta data. Consequently only the first and fourth graphs are considered similar by our algorithm. As mentioned earlier, meta data involved in this step can be any metric collected by Nucleus during the Reconnaissance phase. In our concrete implementation, the number of instructions forming the basic block and the number of calls and their targets are considered. To enable matching among different compilers, the number of instructions is not required to be exactly the same, but rather within a similar magnitude. A measure found to be working decently for the considered cases is ±25% of the size of the bigger block. The number of calls is checked strictly, meaning that there must be the same amount of calls for matching basic blocks, and call targets have to be consistent within one function: If a function signature dispatches calls to the same target from multiple basic blocks, matching only succeeds if the call targets in the basic blocks observed in the unknown code are consistent. Our algorithm to find an isomorph mapping between nodes of two graphs $g_1$ and $g_2$ makes two assumptions. The function has only one entrypoint, and second all nodes are reachable from the entrypoint. Algorithm 1 shows pseudocode of our implementation. It is initially called with the entry points as $n_1$ and $n_2$ and an empty mapping. *suc1* returns the list of descendants of a node $n$ in $g_1$, likewise *suc2* returns a list for the descendants in $g_2$. *Check* implements the before mentioned comparison of meta data. The algorithm performs a depth first search parallel in both graphs. Under the assumptions made, this function is called at least once for every node in the graphs. Already matched nodes are excluded from sN1 and sN2, so that recursive calls either have two unmatched nodes as parameters or $n$ and $f(n)$, resulting in an one to one mapping. When there are no unmatched descendant left, the mapping is tested to ensure $\forall n \in V(g_1).(n_1, n) \in E(g_1) \leftrightarrow (f(n_1), f(n)) \in E(g_2)$. Since all nodes have to be visited the function terminates and returns a non empty list iff $\forall n, m \in V(g_1).(n, m) \in E(g_1) \leftrightarrow (f(n), f(m)) \in E(g_2)$

*3.2.4 Proximity Analysis.* To improve the matches obtained by the constrained version of graph isomorphism (i.e. to rule out further false positives) the interprocedural call graph is analyzed. Specifically, the number of called functions of all descendants of the current function and the (isomorph) candidate function from the signature database are compared. If the numbers do not match, we assume that the currently analyzed function does not fit into the same location of the interprocedural call graph as the candidate function.

In case of equivalence, isomorphism of the graphs of all descendants of the current function all their corresponding descendants from the signature is checked. This can be interpreted as a unique recursive application of the first three steps of the algorithm to all descendants of the current function. Only if this step succeeds, the current unknown function gets assigned the symbolic name from the function contained in the database.

If the last step fails, however, this is due to either an additional or a missing function call in the descendants. Both possibilities imply that at least one function down the call graph has changed on source code level, as different compiler or optimization levels, apart from function inlining, would leave the interprocedural call graph intact.

Note that recursively checking isomorphism of all descendants for more than one level in the interprocedural call graph (i.e. checking isomorphism of descendants of descendants) turned out to be practically not feasible in our implementation.

Any match that passes all tests of the algorithm is assumed to be correct. As we try to avoid false positives (wrongly labelled functions), we require matches to be unique across all candidates from the database. An example where this is problematic are the `strcpy` and `stpcpy` functions from *glibc*. Both functions copy strings, but the former returns a pointer to the beginning whereas the latter returns a pointer to the end of the copied string. These small semantic differences can not be detected, resulting in both functions to look the same to our algorithm. We present a list of all matching candidates to the reverse engineer for manual analysis, if desired.

## 4 EVALUATION

We tested our approach on a diverse set of binaries from different open source projects, that can be seen in Table 5: *nginx*[1], a popular web server, the reference implementation of the interpreter of the *python*[2] programming language, and finally the GNU C library *glibc*[3]. We compare the results of our approach to Diaphora, a function identification toolkit implemented as an extention to the Interactive Disassembler (IDA).

For each step in the evaluation, we generate a signature database from one binary, and then try to match the graphs contained in

[1] https://nginx.org/en/
[2] https://www.python.org/
[3] https://www.gnu.org/s/libc/

| Project | Versions |
|---|---|
| nginx | 1.10.3, 1.12.1 |
| libc | 2.15, 2.19, 2.21, 2.23, 2.24 |
| python | 3.4.0 - 3.6.2 |

**Figure 5: Version numbers of open source projects used during the evaluation**

the signature database to the graphs extracted from the second binary. Note that (part of) our algorithm is used for both steps: first to generate the signature database, and afterwards to match the functions from the database to the candidate functions of the second binary.

We obtain ground truth from symbol data attached to the binaries itself. If a match is found, the label attached to the target function is compared to the symbol name indicated by the signature. If the name is identical, then the match is assumed to be correct. Leaving the symbol data attached to the binary adds no bias to the matching as symbol data is completely disregarded by our algorithm and only serves to verify the results.

As mentioned before, if a function could not be uniquely identified, all possible matches are presented to the reverse engineer. Depending on the demand for unique matches the result is considered a true positive or false negative. In the results of our evaluation, we indicate numbers for both operation modes of the algorithm (unique and non-unique). If the correct label is not among the presented ones, the data point is treated as a false positive.

To quantify the results, we use the common notion of *precision* and *recall*. Precision is the number of true positives $TP$ divided by the sum of true positives and false positives $FP$

$$\text{Precision} = \frac{TP}{TP + FP}$$

whereas the recall rate is $TP$ divided by $TP$ plus the number of false negatives $FN$

$$\text{Recall} = \frac{TP}{TP + FN} .$$

## 4.1 Sample Set Selection

As the central criterion in Algorithm 3.2 for considering two functions similar is graph isomorphism, it is evident that our approach requires a certain degree of complexity of the control flow graphs of the functions during matching.

To develop an understanding of the minimum required complexity, we applied the algorithm successively to one *glibc* and one *python* binary, where in each step the group of functions consisting of the smallest amount of basic blocks was removed. This effectively creates artificial sets in which only the functions with high amounts of basic blocks (and therefore higher complexity) are contained.

Figure 6 shows how the functions are distributed over the number of Nodes. The graph clearly shows that a non-negligible number of functions consists of less than 10 basic blocks. A second interesting observation is that there exist more functions with one or

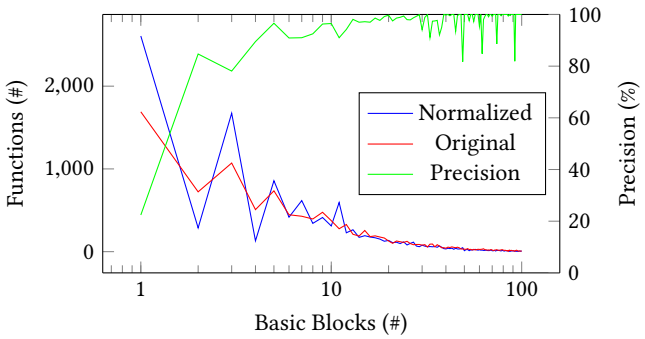


**Figure 6: Distribution of function sizes measured in basic blocks, and precision of the algorithm performed on *python* and *glibc***

three basic blocks as there are with two, and that normalization amplifies this difference. The reasons for this is that for a function to consist of exactly two basic blocks, it either has to contain a loop or do a call. As a call only has one successor in the CFG of the function both basic blocks will be merged into one during normalization. Functions with a small number of basic blocks are very restricted in the diversity of their CFG and matching becomes less precise.Matching of these small functions can therefore not be done in meaningful way using only graph isomorphism.

For our approach we therefore set an artificial lower bound of five basic blocks for a function after the normalization step. All functions that fall below this threshold are disregarded in the evaluation. In our experience, this bound decreases the number of functions that can be handled by the algorithm by about a third, depending on the binary. In the following parts of the evaluation the recall rate is calculated on the base of the number of functions that are large enough. The set of functions with enough basic blocks that can also be found in the target binary is the set of all possible matches.

## 4.2 Normalization Effects

To understand the effects of the (optional) normalization step of our algorithm, we performed matching on all tested pairs of binaries twice, once with normalization enabled and once with normalization disabled.

In Figure 7 we can see the results of matching *glibc* and *python* against all other versions in a pair-wise fashion. Specifically, we matched each binary against each other binary, and computed the arithmetic mean over all obtained numbers. The python binaries were build using gcc version 7.2.0 on optimization level '-O3' and the libc binaries were obtained from ubuntu packages like mentioned earlier.Here we observe that the normalization has no positive impact on the matching. For example, the number of true positive matches (regardless of their uniqueness) decreases by 4.5 percentage points from 48.4% to 43.9% for *glibc* when turning on normalization.

The resulting precision for this test amounts between 95 and 99 percent, with the precision being slightly worse in case normalization was performed.

To understand the effects of normalization when using different compilers and optimization levels to compile the same program,

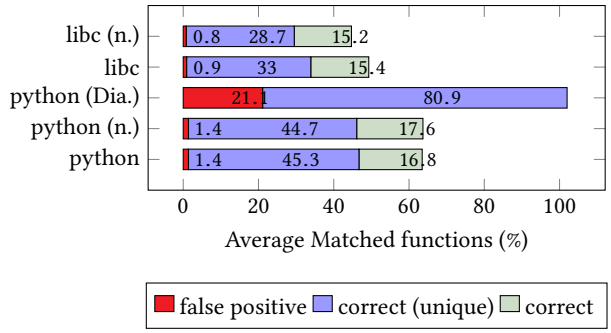


**Figure 7: Sum of false and true positives relative to all possible matches, with unique and non-unique matches separated, (n.) indicates that normalization was active during signature matching. The results of Diaphora are marked with (Dia.). Numbers are arithmetic means of all runs matching one program version to all other versions of the same program.**

we compiled *nginx* and *python* with *gcc* versions 6 and 7 and *clang* version 4.0.1, and with optimization levels 1, 2, and 3.

Figure 8 depicts the results for *python* and *nginx* built with different compilers and options. The numbers indicate that normalization improves the recall rate notably, but also introduces additional false positives. While the recall rate is doubled for comparing binaries compiled with -O1 with their -O2-counterparts, the overall recall rate of 3.5% of the underlying graph matching algorithm is still low.

While Diaphora was able to correctly identify many more functions than our approach, it also produced considerably more false positives. While our approach achieved a precision of 78% comparing functions on optimization levels '-O1' and '-O2', Diaphora only had 31%.

### 4.3 Performance

Runtime was not a focus point for this approach, subsequently the analysis can be rather costly. Matching different versions of the libc took five seconds, while matching the about 19000 functions of the opencv library took about three minutes when running single-threaded on an Intel Core i5-3550M @ 3.10Ghz. The only observed abnormal runtime, where the naïve implementation of graph isomorphism timed out was the implementation of the `strftime` function consisting of a large switch statement implemented by a jump table with 123 entries. This case occurs once within our set of about 34000 different functions, so we conclude that this explosion in runtime is an exception. All other (smaller) `switch` statements were handled seamlessly.

## 5 DISCUSSION & FUTURE WORK

While the results presented in the evaluation look promising, the limitations of the approach can be clearly seen as well. Differences in the compiler seem to have the highest impact to the recall rate. This can be even severe enough to render this approach useless in some circumstances. In the following, we highlight the most common reasons preventing successful graph based matching, and give ideas on how to overcome the limitations of our approach.

### 5.1 Compiler Code Generation Habits

Difficulties to this approach mainly arise from the code generation habits of compilers while translating different aspects of the program.

A `switch` statement for example can be implemented as a binary search tree, or by performing a relative (or absolute) jump based on values taken from a jump table generated at compile time. The method and layout used are dependent on the compiler, the value ranges on which the `switch` operates on, and the optimization level.

*Loop unrolling* is another optimization employed by compilers with severe consequences to the CFG. Here, knowledge about a loop is used to unfold the loop body into a sequence of statements, eventually getting rid of the loop entirely. All these techniques distort the CFG and impede a graph isomorphism based approach to identify functions.

### 5.2 Cross-Architecture Matching

Further problems arise from the availability of instructions for a certain Instruction Set Architecture (ISA). For example, a short `if`—`then`-statement only setting a variable for example can be implemented in different ways. The straightforward way to implement this is to let the compiler emit an explicit check of the condition followed by a conditional jump skipping the body of the `if`-branch. On the other hand, RISC architectures like *ARMv7* provide predicated instructions, which can commonly be emitted by compilers to implement `if`-statements. This constitutes a problem for graph based function matching across different architectures, as one implementation introduces an additional edge to the CFG, whereas the other implementation leaves this jump implicit. Even worse, with the availability of *conditional move instructions* (i.e. `cmovz`) the problem can also occur when comparing code from different compilers or different optimization levels on the same (x86) architecture. Conceivably, this issue could be addressed by adding another normalization step which—depending on the presence of conditional instructions—creates a new basic block containing the corpus of the `if` statement and inserts a new edge into the CFG.

More subtle ways of ISAs to differ are branch prediction and cache coherency. Advanced compilers like *gcc* or *clang* can take these differences into account to create code optimized for the exact platform. This is also a factor preventing machine independent function matching.

### 5.3 Obfuscation Resistance

Another limitation is that no semantic analysis of the code is performed in any way. Functions that differ in the control flow graph but have the same semantics are unlikely to get matched by our algorithm as the normalization steps are insufficient. For example, machine code obfuscation is a common scenario where (typically) semantics preserving changes are made to the CFG. Inserting jumps that are never taken (so-called *Bogus Control Flow*) distorts the CFG sufficiently such that functions do not match any more. These problems can be addressed by improving the normalization and preprocessing step: While matching functions based on their semantics is not possible in the general case, functions without backward edges could be semantically checked for equivalence, as they are guaranteed to terminate.

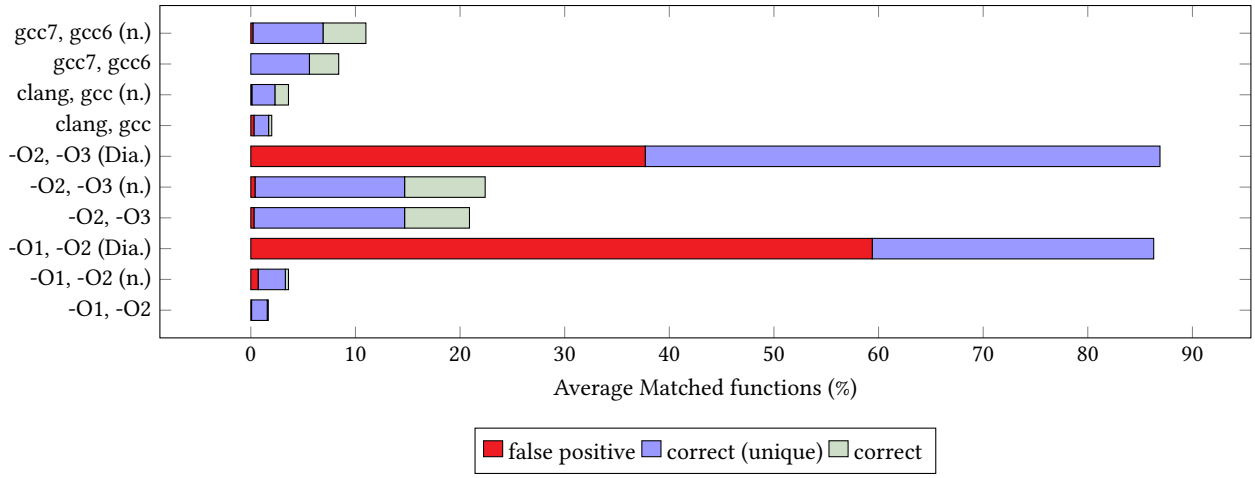


**Figure 8: Sum of false and true positives relative to all possible matches, with unique and non-unique matches separated, (n.) indicates that normalization was active during signature matching. Results of Diaphora are marked with (Dia.) The tests performed compare the effects of using different compilers and optimization levels on normalization. Numbers are arithmetic means obtained while matching *nginx* and *python***

Another improvement can be made by employing a fine grained semantic analysis at basic block level to reveal more subtle differences of blocks that are indistinguishable by other measures. This however would be costly and should only be employed if otherwise there would be exponentially many possible matches, or as a refining step after the matching to resolve non unique matches. One example where this idea could be employed is to solve the problem arising from jump-table-based switch statements containing many entries, that most likely caused a timeout while matching one particular function.

## 6 RELATED WORK

Compiler like gcc and clang produce a deterministic instruction stream to allow reproducibility of software build. Together with the use of calling conventions function entry points have a predictable pattern. Building upon these principals, a byte signature based approach for identifying functions has been created. The first attempt towards identifying library routines in binary executables was published in 1998[**?** ] by Van Emmerik. The method proposed hashes the first few bytes of a function to quickly match byte sequences against known library functions. Before any function can be successfully matched, a pattern database has to be build up. This is done before analyzing the target binary by examining known libraries and object files that contain the functions together with identifying symbols. From these symbols the start of the functions in the binary can be determined and subsequently patterns are derived from the functions. Later, these patterns can be used on unidentified functions during the analysis of a target binary. A famous implementation that follows this idea is implemented in the well known Interactive Disassembler (IDA)[4] and called 'Fast Library Identification and Recognition Technology' (F.L.I.R.T).

Shirani et al. propose *BinShape* [**?** ], an approach that shares some parts with our method: In a first step functions are detected and features about them are extracted. These features include the CFG, instruction level features like the number of calls, as well as general statistical features. Now the approaches diverge, while we do a costly analysis based on the function graph, they propose to use machine learning to generate possible matches. As no source code has been published an evaluation and direct comparison was not possible.

BinDiff[5] has to be noted in the context of function identification as well. It is implemented as a plugin for IDA and employs a wide variety of different measures to identify functions. It builds up a diverse list of attributes per function including callgraph and CFG related ones. Some normalization is applied to the CFG like in our approach, leaf inlining is not[**?** **?** ]. These attributes are then used to find functions across binaries.

Diaphora[6] is similar to BinDiff as it also builds upon IDA and uses a wide variety of measures.

Compared to the mentioned approaches above, the central idea of our method is to use additional normalization steps prior to performing graph based matching, as well as the use of basic block level information during the matching.

## 7 CONCLUSION

While the original goal was to create an algorithm that can match functions between different compiler versions to some extent, the result shows that even for small functions compiler output varies greatly. While the proposed transformations have a positive impact in these circumstances, even doubling the amount of matches in some occasions, the number of functions identified across different compiler versions (amounting about one percent) is rather insignificant. Matching in these cases worked best with *gcc* and *clang* both

[4]https://www.hex-rays.com/products/ida/

[5]https://www.zynamics.com/bindiff.html

[6]http://diaphora.re/

operating on optimization level `-O3`. We conclude that further transformations are required to unify the control flow graphs to further improve matching results.