Received 21 October 2022, accepted 18 November 2022, date of publication 28 November 2022,
date of current version 1 December 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3225100

RESEARCH ARTICLE

BinCC: Scalable Function Similarity Detection in Multiple Cross-Architectural Binaries

DAVIDE PIZZOLOTTO[1] **AND KATSURO INOUE**[2]
[1]Osaka University, Osaka 565-0871, Japan
[2]Nanzan University, Nagoya 466-8673, Japan

Corresponding author: Davide Pizzolotto (davidepi@ist.osaka-u.ac.jp)

This work was supported by the Japanese Society for the Promotion of Science (JSPS) KAKENHI under Grant 18H04094.

ABSTRACT With the undeniable increase in popularity of open source software, also the availability and reuse of source code have increased. While the detection of code clones helps tracking reuse and evolution while dealing with source code, little prior work exists that can be used in binary code. This is complicated by the increased difficulty posed by the compilation transformations.

In this paper, we present a CFG refinement useful to find function-level clones in a fast and scalable way by comparing the high-level structure of multiple disassembled binaries altogether. We are capable of determining if functions belonging to other programs have been copied or reused, even when the processor architecture is different. Specifically, our algorithm consists in the extraction of the various functions flows and the reconstruction of a higher level structure, leveraging architectural differences and allowing efficient comparison in linear time with structural hashing.

We implemented our idea in a tool called BinCC, and analyzed 24 million functions spanning different architectures and optimization levels. Results show that our approach can achieve precision between 91% and 99% within the same architecture and 75% in detecting clones among different architectures, and can also detect the presence of specific library functions inside an executable. Our approach can reach comparable precision of current state-of-the-art learning approaches while being three order of magnitude faster.

INDEX TERMS Code clones, static code analysis, reverse engineering, compilers.

I. INTRODUCTION

Free software has grown in popularity in recent years, and with that also the adoption of this kind of software by companies and its integration inside closed source projects. This popularity is mainly driven by the ease of customization and flexibility rather than economic reasons [1]: code is usually modified, adapted or simply reused and then re-released with a compatible license. With a higher availability of code, another common practice has become copy-pasting and reusing source code in form of small code snippets from online discussion platforms such as Stack Overflow. Aside from the potential license violation, reused code snippets have been proven to be usually harmful [2] or with security flaws that in the original repository have long been patched [3]. Although code cloning has been successful at determining copied function reuse both for legal implications [4] and plagiarism detection [5], its main scope is usually code evolution and vulnerability propagation [6], [7], [8].

Nonetheless, in recent years, several clone detection tools targeting lower-than-source-code have been developed. These are mainly driven by a normalization step performed by the compiler, rather than the lack of availability of the source code. These works have been targeting Java Bytecode [9], [10] or LLVM IR [11].

Different motivations can be found when working exclusively with binary code: we already cited the license violation in closed source software [12], [13]. Additional scenarios may involve detecting the presence of a vulnerable function in proprietary software [14], [15], analyzing the evolution of

The associate editor coordinating the review of this manuscript and approving it for publication was Claudia Raibulet.

closed source software or the evolution of compiler transformations, and even aiding the disassembly when a known function snippet is detected.

While existing works in the field are mainly focused in malware detection, the main motivation for our work is software evolution and propagation in binary files, even in case of security-related issues (e.g. bug propagation in release of proprietary software). For this reason, the focus points of our research are the ability to analyze multiple executables at the same time and detect the propagation of a given function, the ability to perform such analysis in a reasonable time for any amount of given executables, and the ability to work amongst different architectures, even lesser known.

In order to address these problems, our approach is based on the idea of generating a common structure between the compiled code in different architectures in order to leverage compilation differences, similar to how decompilation helps normalize source code differences [16]. In particular, starting from a Control Flow Graph (CFG), we generate a higher level structure using structural analysis. This structure allows efficient comparison in linear time, as opposed to the exponential time required by a subgraph comparison. This structure has the additional property of normalizing the various architectural differences, allowing cross-architectural comparison. Thanks to its linear time scalability, our approach can be used to efficiently analyze multiple binaries altogether, outlining the evolution and propagation of code among huge codebases such as the LLVM project or the GNU toolchain, while existing works are usually limited to a pairwise comparison [17], [18].

The main novelty of our work is the following:

- A novel structural analysis designed specifically for clone detection.
  Unlike previous work in decompilation (e.g. [19]), our analysis can not emit `goto`s and thus must sacrifice some correctness. For this reason, we had to develop a novel analysis with different detection rules for converting a CFG. This new analysis is described in detail in Section III-D.
- A novel comparison method to efficiently detect structural clones among multiple binaries.
  Comparing two CFGs requires an exponential algorithm [20] and even the average function length of 40 nodes is intractable. While previous works have based their comparisons on statistical properties [14] or dominator trees [21] we based our comparison on tree hashing, allowed by our structural analysis.
- An evaluation of clone detection across different CPU architectures.
  Previous work using structural or semantic analysis always targeted trivial binaries such as Java Bytecode [16], [21] or has been limited to the same architecture [22]. Whereas cross-architecture clones have been researched with deep learning methods, we propose a structural analysis algorithm that works in binary code and does not require a training step.

The paper is structured as follows: Section II presents the State of the Art in code cloning for binaries and binary analysis. Section III explains in detail how the entire analysis is done, including the structural analysis algorithm and the final comparison. Section IV evaluates our approach across varying architectures. Section V describes some limitations and threats to the validity of our work, and finally Section VI closes the paper.

II. RELATED WORKS

Given the intrinsic difficulty at analyzing binary files and the high amount of information lost during the compilation process, the literature in clone detection dealing with compiled executables is more scarce compared to the one dealing with plain source code. The most complete and, to our knowledge, only existing analysis aimed at finding clones in binary files is the one performed by Sæbjørnsen et al. which uses a semantic approach based on vectors containing the instruction sequences [22]. This analysis is itself an extension of the one developed by A. Schulman [23].

In the matter of license violation, on the other hand, Hemel et al. performed three different types of analyses based on string, data compression and binary deltas analysis [24]. Both these approaches have been tested only on executables or libraries within the same architecture, although the string analysis of Hemel et al. could technically be applied also to different architectures.

The structural reconstruction, instead, has been studied in-depth by Engel et al. [25] and later refined by Brumley et al. [26] and Yakdan et al. [27]. The latter two, however, studied an approach more akin to decompilation and semantic preservation while we care more about having the same reconstruction in different binaries or architectures rather than a correct reconstruction.

In the software security field, several tools are available to check for similarities between binaries. In particular, BinDiff [17] and DarunGrim [18] present a structural analysis that use CFG isomorphism and basic block matching. BinSlayer [28] and discovRE [14] improve the existing work by providing faster CFG matching. However, these studies are aimed at finding bugs by analyzing CFG properties, while we refined even further the analysis in order to transform the CFG into a tree and have linear time comparison. This allows us to process thousands of functions in seconds. In contrast to structural analysis, some tools present a detection based on semantic properties. It is the case of BinHunt [29] using symbolic execution and BinJuice [30] that extracts the semantic representation of basic blocks. While all these works are essentially single-architecture, Pewny at al. provided a cross-architectural one by using a semantic representation of basic blocks [15], while, in contrast, our tool uses structural similarity. In addition to these tools, BinSim [31] can support obfuscated binaries.

In recent years, with the prevalence of IoT devices, cross-architectural analysis of binary files has been the scope of work such as BinGo [32] and CACompare [33] that use the signature of a function to perform comparisons. Hu et al., in particular, investigates also the impact of different compilers and optimization levels [34].

Recent works have also started introducing Deep Learning approaches in order to detect similarities between binaries [35], [36]. While some of these were focused just on graph properties [37], recent approaches are more similar to Natural Language Processing [38], [39], [40], [41].

The main difference between our approach and Machine Learning-based ones is the lack of requirement of a training step: building and retrieving a training dataset is already challenging for some dominant CPU architectures [38], [41], and it may be almost impossible in case of obscure or proprietary architectures. Our approach implementation, instead, currently supports 20 architectures, and additional ones can be added by writing a few lines of code, provided a disassembler is available. An additional downside of the learning based approaches is the runtime: several works such as InnerEye [38] and DeepBinDiff [39] can reach high precision, but require an enourmous amount of time to run, that usually hampers scalability. In our approach, instead, the runtime is completely dominated by the disassembly time and is orders of magnitude faster than the aforementioned approaches.

III. APPROACH

This section presents our approach for detecting function clones in a binary file. Our idea consist of generating a tree-like representation of each function, starting from a CFG, that may represent the original source code structure. Then, efficiently checking for clones in the subtrees of these representations. This come from the idea that, even for different architectures, the duplicated code once compiled should retain the same structure in both architectures.

However, directly comparing a graph as the CFG is not trivial, as it is known to be a NP-complete problem [20]. Comparing two CFGs with an exponential algorithm, for every pair of function we want to process, is thus unfeasible.

For this reason we use a structural analysis step, to reduce a CFG into a tree, more suitable for analysis. This step is akin to a decompilation, but it differs from it in the sense that semantic correctness is secondary as opposed to reconstruction completeness, allowing us to remove CFG edges.

A. OVERVIEW

The overview of our approach can be seen in Figure 1. In this figure, several binaries labeled as *Binary 1*, *Binary 2* and *Binary 3* are analyzed in order to find cloned functions between them. We can see from the figure that our approach requires one or more binary files as input, and produces a single output: this output contains clone classes (clone sets) [42] which indicate cloned functions. Each clone class represents functions implementing the same behavior: two or more functions are reported as belonging to a clone class along with their name, original binary name and cloned basic blocks. Our approach compares all functions of all binaries together and is based on three main steps namely Structural Analysis, Comparison and Semantic Analysis.

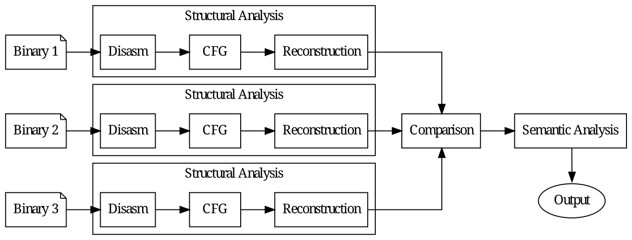


FIGURE 1. Overview of the binary analysis.

Structural Analysis is used to retrieve a structural representation of every binary function. With those representations, the Comparison step generates the clone classes, and the Semantic Analysis further filters those clone classes from false positives. A detailed description of the various steps in Figure 1 is the following:

Disasm
In this step, the original binary code is disassembled and, for every function, a list of tuples `<offset, mnemonic, arguments>` is obtained. This step is performed once for every binary.

CFG
In this step, the CFG of each function is retrieved, starting from the list of statements. Additionally, these CFGs are refined and slightly modified as explained in Section III-C. This step is performed for every function retrieved during disassembly.

Reconstruction
This step is the core of our approach: here each CFG is transformed into a representation of the original function using nested high-level structures (e.g. If, While, Do-While). These nested structures can represent the function in the form of a tree thus allowing constant time comparison, unlike a CFG. Our analysis, however, does not completely preserve the program structure. In fact, our approach modifies the CFG in case the analyzed function does not have structured control flow (e.g. loops with `break` and `continue` statements). For this reason, the algorithm is explained in detail in Section III-D along with reconstruction rules for each CFG pattern.

Comparison
The results of the reconstruction are compared altogether and if any duplication or match is found it is reported as output. Comparison is done using an hash-based approach, by searching for hash collisions while linearly processing all functions. Further details on this step are explained in Section III-E.

Semantic Analysis
Finally, in this step potential clones reported by the comparison step are checked for semantic consistency. A different approach is used based on whether the potential clones belong to the same

architecture or not. This step is described in detail in Section III-F.

B. DISASSEMBLY

The first step of the entire analysis is the disassembly, requiring as input the original binary file, either as executable, library, or object code. Given that a binary file is just a collection of bytes interpretable as machine code, data, or comments, the purpose of this step is taking as input the aforementioned binary file and providing as output assembly instructions required by the subsequent steps. An assembly instruction is a pair `<mnemonic, arguments>` where *mnemonic* represents a particular instruction to be executed by a CPU, taken from a set of instructions composing the CPU Instruction Set Architecture (ISA), and *arguments* is a sequence of elements passed as arguments to the *mnemonic*. Additionally, ensuing steps of our analysis require assigning a unique *offset* to each instruction. This *offset* is generally calculated as the number of bytes from the beginning of the file until the instruction, and must be coherent with the jump targets: if an instruction performs a jump to another one, the argument of the jump must be the offset of the target instruction.

We can thus say that, for the purpose of our analysis, an assembly instruction is a tuple `<offset, mnemonic, arguments>`, and an example of this can be seen in the following code:

```
0x601 cmp dword [var_4h], 0
0x605 je 0x60E
0x607 mov eax, 0
0x60C jmp 0x613
0x60E mov eax, 1
0x613 pop rbp
0x614 ret
```

In this code, each line represents an instruction. The first number of each line is the offset, followed by a space and the mnemonic. Everything else on each line are the mnemonic arguments.

The expected output from this step is a list of function names, and for each function a list of assembly instructions. In our implementation we relied on external tools in order to perform the disassembly, as such, in order to keep generality, subsequent steps in our analysis assumes this list of assembly statements as plain strings in Intel syntax, as presented in the previous example.

C. CONTROL FLOW GRAPH

The purpose of this step is the transformation of the list of instructions obtained in the Disasm step into a suitable representation in form of a CFG, required for the structural analysis. This step is performed for every function obtained in the disassembly step. A CFG is a directed graph $\mathcal{G} = (V, E)$ where every node $v_k \in V$ represents a basic block and every edge $e_{i,j} = (v_i, v_j) \in E \wedge v_i, v_j \in V^2$ represents a possible movement from the basic block v_i to the basic block v_j. This kind of graph is effective at presenting the flow of the program and is the starting point for our structural analysis.

Given the relatively easy task of transforming a list of statements into a CFG, we are not going to present its implementation here. The only requirement is to have the list of conditional and unconditional mnemonics for each ISA.

Additionally, after retrieving the CFG, a refinement step is performed, consisting of the following actions:

Single exit
If multiple exits for the analyzed function are found, a new single exit is created. This step is required to preserve consistency with some patterns defined later in Section III-D.

Dead nodes removal
Dead basic blocks, unreachable from the root, are removed from the resulting CFG. These basic blocks are resulting from indirect jumps, i.e. jumps to a dynamically known address which are unsolvable at static time and thus ignored during the CFG reconstruction [43], [44].

D. RECONSTRUCTION

The refined CFG with a single entry and a single exit obtained in the previous step is refined in this step. In particular, we aim at iteratively reducing CFG portions into high-level structures until a single node is left. This node will represent the entire function, and recursively contains the various structures composing it. This representation two major advantages over a plain CFG:

- Loops are removed, and the resulting output is a rooted tree. The resulting tree nodes are labeled by type, allowing faster comparison through hashing. Comparing a CFG instead requires exponential time [20].
- Minor differences between the `x86_64` and `aarch64` CFGs are leveraged.

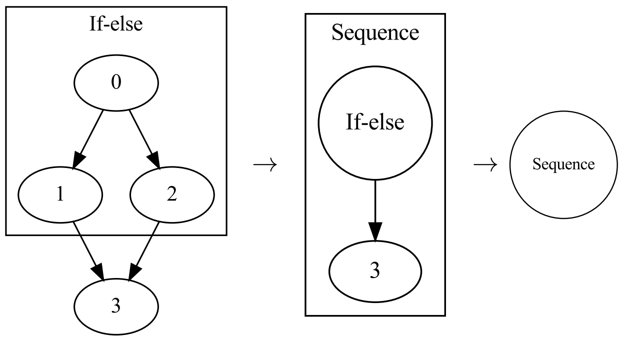


FIGURE 2. Example of the Reconstruction step. The nodes 0, 1 and 2 of the CFG are removed and replaced with a node labeled as If-else structure. The new If-else structure and node 3 can then be replaced with a Sequence structure. When only one node is left the algorithm terminates.

An example of the reconstruction can be seen in Figure 2, with its interpretation in tree form on Figure 3. On the left of the Figure, the basic blocks 0, 1 and 2 forming an *If-else*

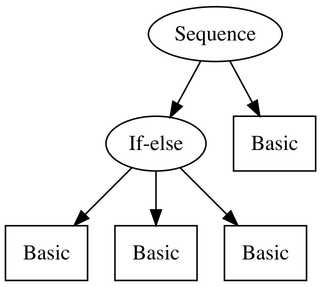


FIGURE 3. The CFG of Figure 2 represented as tree after reconstruction.

structure have been reduced. Then, in the center, the newly created *If-else* structure and basic block 3 are reduced into a *Sequence*, thus terminating the algorithm, as shown on the right. This analysis reduces CFG portions to the following structural types: *Sequence*, *Self-loop*, *If-then*, *If-else*, *While*, *Do-While*, *Switch*, *Optimized If*.

Algorithm 1 Reconstruction Main Loop

```
inputs : A CFG $\mathcal{G} = (V, E)$
output: A single node representing the nested structures. Nil
        if the procedure failed
Function Reconstruction($\mathcal{G}$):
    $sccs \leftarrow$ FindStronglyConnectedComponent($\mathcal{G}$)
    foreach $scc \in sccs$ do
        TransformNaturalLoop($\mathcal{G}$, $scc$)
    while $|\mathcal{G}| > 1$ do
        $list \leftarrow$ postorderDFS($\mathcal{G}$)
        while $list \neq \emptyset$ do
            $node \leftarrow$ pop($list$)
            $\mathcal{R} \leftarrow$ reduce($node$)
            if $|\mathcal{R}| > 0$ then
                Let $r$ be a single node containing $\mathcal{R}$
                $\mathcal{G} \leftarrow (\mathcal{G} \setminus \mathcal{R}) \cup r$
                foreach $(v_i, v_j) \in E \wedge v_i \notin \mathcal{R}$ do
                    if $v_j \in \mathcal{R}$ then
                        $v_j \leftarrow r$
                break
        if not modified $\mathcal{G}$ then
            return nil
    return $\mathcal{G}$
```

Algorithm 1 presents the reconstruction procedure. In the *Reconstruction* function, the first three lines are used to modify Natural Loops, loops with more than one exit, and are explained in Section III-D6. In the remaining part, we can see that the outer `while` is the iterative step, running until the graph is composed solely by one node. For each iterative step, a post-order depth-first visit is performed. The reason for the post-order visit lies in the fact that while processing a node, every possible descendant of it has already been given the possibility to be reduced beforehand. Then, the inner `while` attempts to reduce each node of the post-order visit to a known structure by calling the *reduce* function on each node.

The *reduce* function takes a node as input and returns a set, called $\mathcal{R}$, containing the original nodes composing a particular structure. Reductions are tried in the following order: *Self loop*, *While* and *Do-while* altogether, *If-then*, *If-else*, *Sequence*, *Switch* and *Optimized If*. If the reduction succeeds, firstly every node composing the region is removed from the CFG and a new structural node named r is added. Then, for each edge $e_{i,j} = (v_i, v_j)$ where v_i does not belong to the reduced region, the target v_j mapping to a component of the region itself is remapped to r. The edge becomes $e = (v_i, r)$, effectively replacing a region with a single node.

Lastly, the inner loop is terminated, given that the post-order visit is now invalidated having the CFG been modified. The outer `while` generates a new post order visit and the process is repeated until a single node is left. If, instead, the list is processed in its entirety without any modification to the CFG the procedure terminates with a failure state. We now explain how every region has been reduced, excluding self-loops which are trivial.

1) IF-THEN RESOLUTION

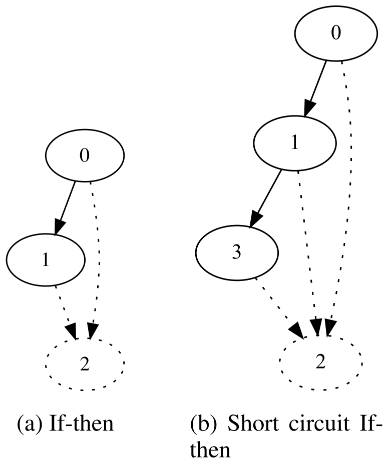


FIGURE 4. If-then and Short circuit If-then. A sequence of nodes composed of conditional jumps with the target being fixed (2) can be considered as If-then structure.

We can see in Figure 4, on the left, the representation of a minimal *If-then* structure as it would appear on a CFG. Solid lines represent the nodes that will be reduced. Note that in case an *If-then* structure has more complex logic in the *then* node, this has already been reduced in previous iterations. In order to reduce a node as an *If-then* region, the following conditions must hold:

1) The current node has two children, the *then* node and the *next* node.
2) The *then* node is a node with a single predecessor and a single successor.
3) The successor of the *then* node is the *next* node.

If all these conditions are satisfied, the current node and the *then* node are transformed into an *If-then* region, with the *next* node as its successor. In Figure 4, on the right, we can instead see the CFG for an *If-then* with a short-circuit evaluation. Recall that short-circuit evaluation is the semantic of a boolean expression where some arguments are not evaluated if the truth value of the expression has already been established, represented in this example by the edge $e_{0,3}$. This kind of *If-then* is automatically resolved iteratively using the previously defined rules, generating a nested *If-then* region where the *then* node is another *If-then* region. In our implementation we flattened these nested nodes into a single *If-then* keeping the first $n-1$ nodes as the various conditions and the last node as the actual *then*. Moreover, the number of conditions are not considered in the comparison phase, so a short circuit *If-then* is reconstructed identical to a normal one.

2) IF-ELSE RESOLUTION

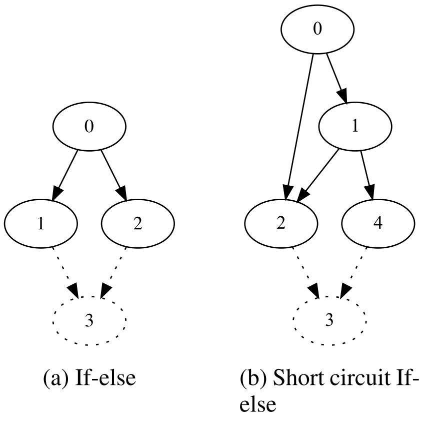


FIGURE 5. **If-else and Short circuit If-else. Two different paths sharing a node (0) and a sink (3) can be reduced as an If-else structure. In principle, the two paths must be disjoint, except in the case where every node of a path connects to the same node of the other, indicating a short circuit If-else.**

In Figure 5 on the left, the minimal structure of an *If-else* is presented. Exactly like the *If-then* resolution, the following set of conditions must be verified in order to reduce a node as an *If-else* node:

1) The current node has two children, the *then* node and the *else* node.
2) The *then* node has a single predecessor and a single successor.
3) The *else* node has a single predecessor and a single successor.
4) The successor of the *then* node is the same of the *else* node.

If all these conditions are satisfied, the current node, *then*, and the *else* node are merged into an *If-else* region with the successor of *then* as successor. It is important to note that deciding which node is *then* and which node is *else* is not trivial and has some consequences, however, this will be discussed in Section III-E when dealing with the comparison. Additionally, unlike the *If-then* structure, the short-circuit version of the *If-else* presented on the right in Figure 5 is not resolved automatically using the previous rules, given that the rule number 3 is violated by the short-circuit. It is thus required to implement a routine that tries to descend into the *then* subtree, asserting that every successor is either another *then* node or the *else* node. The new rule 3 instead will ensure that every predecessor of the *else* node is either the current node or one of the *then* nodes. Also in this case, short-circuit version and normal one are treated equally in the comparison phase.

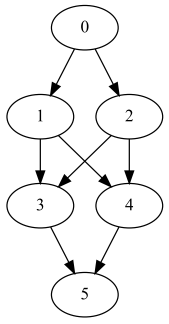


FIGURE 6. **Optimized If structure. This structure is similar to an If-else, but the two paths are not disjoint.**

An interesting case can be seen in Figure 6. The figure represents a particular *If-else* where the *then* branch can arbitrarily jump to the *else* one. Although impossible to realize in code without `gotos`, this construct is often emitted by the compiler to handle exit conditions and failures. Unfortunately, a compiler can generate arbitrarily long *then* and *else* branches, resulting in endless different between-branch jump possibilities. This means that either we ignore some of these structures, increasing the failure rate for the structural analysis, or we use the same label for structures that are not the same, sacrificing comparison correctness. In our study, we chose the first approach, as we further discuss in Section IV-A. In particular, we detected only the most basic compiler-generated patterns like the one in Figure 6 and labeled them as *Optimized If*.

3) SWITCH RESOLUTION

A more complex variant of the *If-else* is the *Switch*. These structures, after compilation, are usually implemented with a jump table and thus require particular care to be detected statically. Fortunately, several algorithms for recovering statically these tables exist in literature, and the task is thus delegated to the disassembler [45].

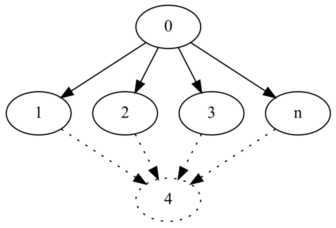


FIGURE 7. **Switch structure. Any sequence of disjoint paths from a node (0) to a sink (4) with more than two paths can be reduced to a Switch structure.**

In the CFG, switch thus appear as in Figure 7 and can be easily detected as its root node has a number of edges higher than 2. If all these nodes point to the same successor, they are reduced as *Switch*, otherwise the successors should be refined first.

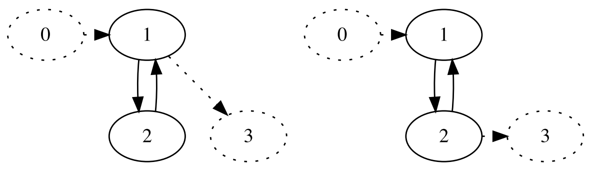
4) SEQUENCE RESOLUTION

Sequences do not present particular cases, so it is sufficient that the current node and its successor satisfy these conditions to merge them into a 2-nodes sequence:

1) current node has a single exit
2) successor has a single entry and a single exit

Additionally, if the current node or the successor is already a sequence, the newly created one is not nested but appended to the existing ones.

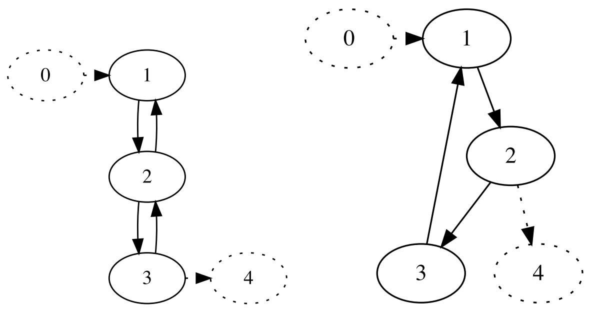
5) WHILE AND DO-WHILE RESOLUTION



(a) While (b) Do-While

FIGURE 8. While and Do-While loops structures. A path of length 2 starting from and ending in the same node can be considered a loop. If the loop entry and exit are from the same node, this loop can be reduced as a while structure, otherwise as a Do-While structure.

While and *Do-While* loops are conceptually similar, as can be seen in Figure 8 with the former on the left and the latter on the right, the only difference being the exit node. While these loops can be found quite easily using the Tarjan's Strongly Connected Component (SCC) algorithm, particular care must be taken when dealing with nested loops, where the head of a loop is also the tail of the other one, as shown in Figure 9 on the left. Specifically, the SCC cannot be blindly used to determine the exit of a loop, given that in case of nested loops both the inner and outer loop have the same SCC. Additionally, unlike *While* loops, *Do-While* constructs can have a minimal form composed of three nodes that can not be reduced to two nodes using *Sequence* rules, and requires an ad-hoc reduction. This form is shown in Figure 9 on the right.



(a) Nested Do-While (b) Do-While with 3 nodes

FIGURE 9. Nested Do-While and minimal 3-nodes Do-While. These are two particular cases of Do-While structure that require additional care: (a) for the correct identification of loop entry and exit and (b) because the combination of Sequence and Do-While detection rules can not detect this particular case.

6) TRANSFORM NATURAL LOOP

In case the original source code had a loop with *break* or *return* statements, in the CFG this loop could potentially have multiple exits. In order to effectively reduce a loop to a minimal 2-nodes variant, a single exit is required and the first three lines in Algorithm 1 are used to detect and remove these edges from the CFG. In fact, our analysis aims at reconstructing only the original structure and thus these loop-breaking statements are redundant. Given a CFG $\mathcal{G} = (V, E)$ and a particular set of exit edges $EX(k)$ from a SCC k as

$$\begin{aligned} EX(k) = \{\forall e_{i,j} \in E \mid v_i \in V \wedge \\ v_j \in V \wedge \\ SCC(v_i) = k \wedge \\ SCC(v_j) \neq k\} \end{aligned}$$

where $SCC(k)$ is the SCC index, the natural loop resolution is run for the SCC if $|EX(k)| > 1$, meaning that more than one exit edge exists for that SCC. From here on, given $e_{i,j} = (v_i, v_j)$ such that $v_i \in EX(k)$ is an exit edge, we call exit node the node v_i and target the node v_j. Asserted that the number of exit nodes is always greater than one, two possible types of natural loop exist:

Single Target
This type of natural loop is generated by *break* statements without any kind of cleanup, like `if(condition)break;`. If an exit node has a higher number of predecessors than the others it is kept as the real exit, with the assumption that this loop would be a *While* loop. Otherwise, the first exit node encountered in a Depth-First Search (DFS) is kept. Although this could be the wrong one, recall that we are focusing on consistency between different programs rather than decompilation correctness. Every edge not belonging to this exit is removed from the CFG.

Multiple Targets
In addition to the single target, *if* constructs with additional logic before the *break* keyword generate a different exit target. Moreover, also *return* statements generate an additional target. While the latter can be easily spotted by searching a jump directly to the function exit, we decided to keep only the target having the longest path $d(v_j, v_k)$ where v_j is the target, v_k the function return node and $d(v_j, v_k)$ the minimum distance between the two nodes. Every edge pointing to the wrong target is removed, then the natural loop is resolved in the same way as the Single Target one. Note that unlike the Single Target, this approach could generate orphan nodes.

After the removal of these edges from the CFG, the loop can be reduced to the minimal form of the *While* or the *Do-While* presented in Section III-D5 with the iterative step of Algorithm 1.

E. COMPARISON

After completing the reconstruction step, the output graph can be represented as a tree of structures, similar to the example shown in Figure 10. In order to efficiently compare all the possible subtrees generated by the reconstruction of every function, we use Locality-Sensitive Hashing (LSH). Specifically, we choose a hash function such that the collisions between trees with the same structure are maximized. It is of vital importance to carefully design this function: we want to ensure an hash collision for structures that are *similar*, not *identical*. In fact, the functionality and high level structure of two functions may be the same, but they may use basic blocks or jumps with different offsets. For this reason we hash only the node type, avoiding the basic block offsets, recursively.

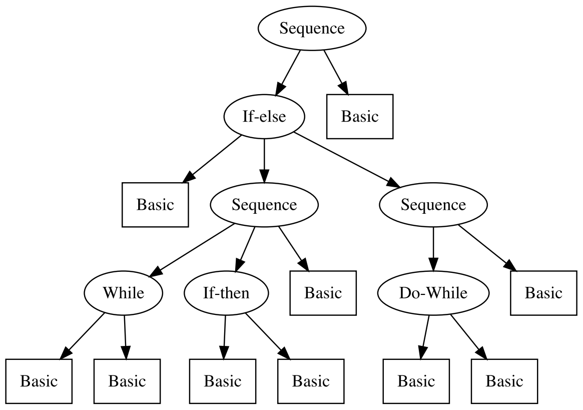


FIGURE 10. **Tree resulting from the reconstruction.**

In our implementation we used as hashing function f_h the public domain FNV-1a [46] and implemented the function $h(t)$, in order to hash a subtree, shown in Equation 1.

$$\begin{aligned} h(t) &= f_h(type(t)) \circ h(c_0) \circ h(c_1) \cdots \circ h(c_n) \\ &\forall c_i \in C_t \\ &\text{where } C_t \text{ is the set containing the children of } t \end{aligned} \tag{1}$$

We can see that the hash of a node t is calculated by composing the hash of the current node type with the hash of all its children. The node type is a unique value assigned to each structure type. If two nodes t_1 and t_2 are both the same structure (e.g. Sequence, If-Else, etc.) $type(t_1)$ and $type(t_2)$ have the same value.

We apply this function to every reconstructed function of each binary. If the tree depth of the current node is higher than a threshold θ, it is written into a hash table data structure, with $h(t)$ as key and t as value. Upon iterating all the keys of this hash table, if a key contains more than one value, the key itself represents a clone class and all its values represent the various clone snippets belonging to this class.

Considering that f_h is a constant time operation, the complexity of $h(root)$, where *root* is the root of each reconstructed tree, is linear in the number of tree nodes. According to our reconstruction rules, a reconstructed tree can not have more nodes than basic blocks, implying that our approach has a comparison step with linear complexity in the total number of basic blocks retrieved from all functions.

We can see that hashing is not based on the statements contained inside a specific structure, but only on the structural shape itself. This enables the comparison between different architectures at the cost of increased false positives ratio, in case two pieces of code are structurally similar but perform different actions. Being thus the hashing distance based solely on the structural shape, we use the threshold θ as variable to control the ratio between false positives and negatives: a lower θ will match structures with few nested nodes that may be shared by code performing different operations, while a higher θ will require a more unique structure but may miss some matches.

We do, however, account for children's order when calculating the hash; it is thus easy to see why in Section III-D2 it was important to clearly determine the *then* node and *else* node deterministically.

F. SEMANTIC ANALYSIS

The comparison presented in Section III-E is sufficient for finding clones, however, its calculations are entirely based on the structure of binary code. This can present a situation where two binary functions are reported as clones because they share the same structure, albeit having different opcodes thus resulting in different functionality. For this reason, in this section we present an additional refinement step to our approach, that should run on the comparison results, in order to filter some false positives that happen to have the same structure, but different opcodes.

This refinement is based on the idea that to every instruction we can assign a number, representing the amount of times that particular opcode appears in the function over the total number of opcodes in the function, thus creating a frequency vector. Then, a function to measure the similarity between two vectors, can be used to measure the degree of similarity between two frequency vectors. If this value is greater than a threshold, the two basic blocks can be considered the same. We run this comparison on the (unordered) list of opcodes composing the basic blocks of two matching reconstructed tree, similar to the one in Figure 10. Moreover we do not consider the opcode parameters: doing so would overspecialize our comparison and match only identical functions. In our implementation we used cosine similarity as similarity function: given two frequency vectors A and B, their similarity is given by $S_{AB} = \frac{A \cdot B}{\|A\|\|B\|}$

Naturally, if the architecture between the potential clones is different, their opcodes would not match and their similarity will always be zero. For this reason, in case of different architectures, instead of using the opcodes directly we assign them to a "family of operation" and determine the frequency of each opcode family instead of the opcode itself. For example, the `jmp` opcode for `x86_64` and `b` opcode for `aarch64` are both assigned to the `JUMP` family, and the similarity calculated on the frequency of the `JUMP` family, instead of the frequency of `jmp` or `b`. In total we divided opcodes into

30 different families. For space reason, we do not report them here, but the full list can be found in the repository referenced in Section VII.

Note that, unlike the comparison approach presented in Section III-E, the semantic analysis presented in this section is performed by comparing two potential clones at a time. This means that, to compare n functions, $\mathcal{O}(n^2)$ operations have to be performed, compared to the $\mathcal{O}(n)$ required by the structural analysis. This explains why, in our approach, the semantic analysis is used only after gathering a list of potential clones from the structural one, and its higher run time is confirmed by the experimental results we provide in Section IV.

IV. EVALUATION

We implemented our analysis in Rust under Linux, however, we are able to analyze the binaries for multiple operating systems and 20 different architectures. The implementation of our approach is named BinCC, and is openly available on GitHub.[1] In this evaluation, we target mainly the `x86_64` and `aarch64` architectures, being the dominant architecture for the Desktop and Mobile market respectively.

As mentioned in Section III-B we depend on an external disassembler to retrieve the list of functions and statements. In our implementation we used the open source tool *radare2*,[2] version 5.7.4, whereas every other step is independent of external tools. Despite using *radare2*, we do not depend exclusively on it: in fact any disassembler could be used, provided that a list of statements like the one in Section III-B is returned as a string.

In order to evaluate the effectiveness of our approach, we want to address the following research questions:

- **RQ1**$_{completeness}$: How many functions are successfully converted to a single node representation?
- **RQ2**$_{correctness}$: How precise is our tool at detecting function clones across different architectures?
- **RQ3**$_{use\text{-}case}$: Is our tool able to find function reuse from libraries in a real-case application?
- **RQ4**$_{performance}$: How performant is our tool, varying the input executable size?

The first research question, **RQ1**, is meant to determine the amount of functions that can actually be reconstructed, across different architectures and optimization levels. This can be seen as a measure of the goodness of our structural analysis algorithm, given that functions that are not reconstructed can not be compared with hashing. The second research question, **RQ2**, is meant to measure the precision of our detection when checking for clones in binaries within the same architecture or different architectures. Additionally, the threshold parameter θ for the LSH function defined in Section III-E is evaluated here and our approach is compared against existing state-of-the-art works. The third research question, **RQ3**, is meant to be a use-case evaluation, measuring the precision at detecting function reuse with binaries and libraries resembling more a real-case scenario. Finally, the last research question, **RQ4**, is used to measure the time required by the analysis and the scalability of our approach.

In order to ensure an evaluation as close as possible to the final use-case, while still guaranteeing a correct evaluation, in this study a different dataset was used for each research question. A summary of the various datasets can be found in Table 1, while details on their creation are listed in the respective Research Questions.

TABLE 1. Statistics and features of the dataset used for each Research Question. Because RQ3 uses publicly available real-case binaries, the optimization level and strip status are not known.

RQ	Binaries	Functions	Stripped	Optimization
RQ1	11211	23953469	yes	O0, O2, Os
RQ2	216	40907	no	O2
RQ3	800	276485	Unknown	Unknown
RQ4	1934	1434790	yes	O2

A. RQ1: COMPLETENESS

In order to answer RQ1 we used as dataset a collection of binaries publicly available on Zenodo [47]. These binaries come from multiple open source projects of different scopes and are compiled using different optimization levels. As reported in Table 1, these binaries are stripped. We used binaries compiled using GCC for the `x86_64` and `aarch64` architectures, divided into `O0`, `O2`, `Os` optimization levels.

For the evaluation, we ran the reconstruction on every function and checked if the output was effectively a single node. Input functions already composed of a single node are not considered in this evaluation.

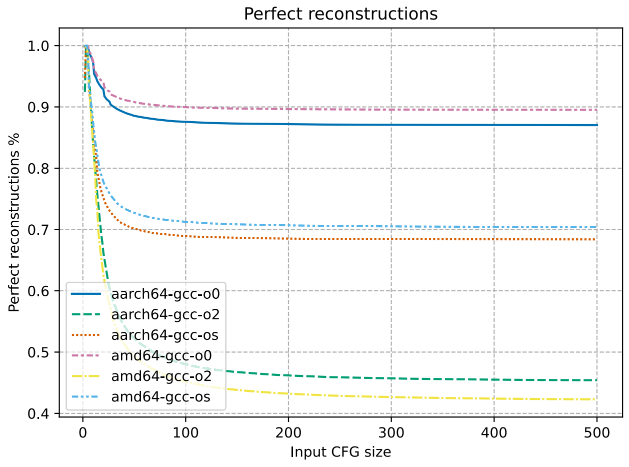


FIGURE 11. Perfectly reconstructed functions on stripped binaries, excluding trivial ones.

Figure 11 shows the percentage of correctly reconstructed functions over the total, for a given CFG input size. The immediate result we can note is that the higher the

[1]http://github.com/davidepi/bincc
[2]https://rada.re/r/

optimization level, the lower the chance our algorithm is able to correctly perform the reconstruction. This is somewhat expected, as higher optimization levels introduce additional compilation patterns that are more difficult to map to the structures we defined in Section III-D. In particular, we analyzed the failed reconstructions and determined that most failures are due to variations of the *Optimized If* defined in Section III-D2. With high optimization levels, the compiler enables jumps between the various if-else branches, generating complex branching structures that cannot be easily categorized.

Differences between the two architectures instead are minor: the reconstruction in `aarch64` is slightly less precise compared to `x86_64` in both `O0` and `Os`, but not in `O2`. We could not determine the reason of this difference by looking at the functions, and we can only assume it is due to the disassembler being more proficient with `x86_64`.

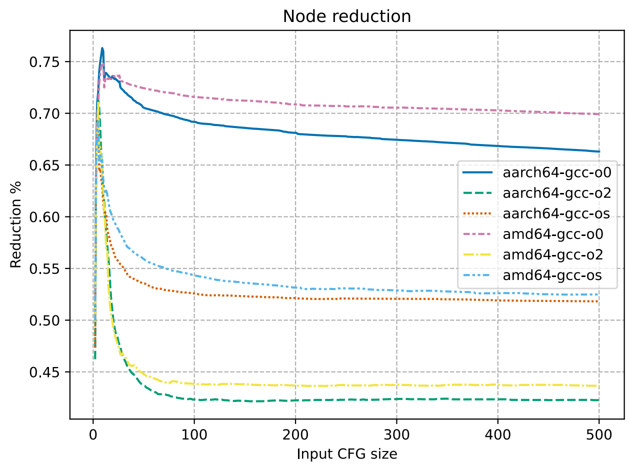


FIGURE 12. **Amount of reduced nodes based on the original CFG length.**

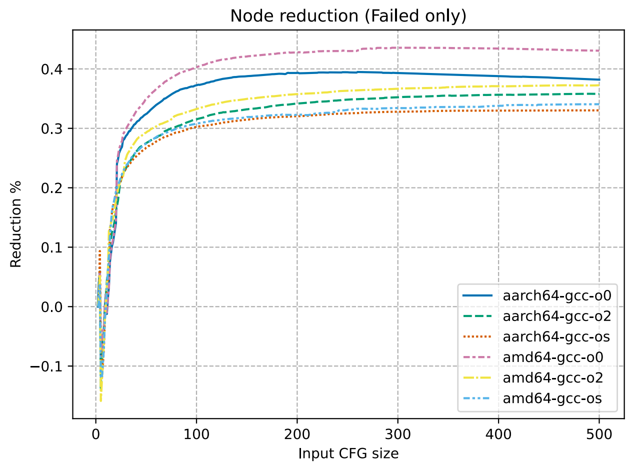


FIGURE 13. **Amount of reduced nodes, considering only failed reconstructions.**

Figure 12 and Figure 13 show the reduction percentage in all functions and the reduction percentage in failed reconstructions only. These are meant to represent how much smaller a reduced function is, compared to the original counterpart, considering the number of nodes before and after a reduction. This is very important, as in case of failed reductions the comparison may still be possible if the number of nodes is sufficiently small.

In fact, the maximum amount of nodes we found is 4202 for the `aarch64` architecture and 9424 for `x86_64` with an average of 40 nodes per function. In these cases, the CFG analysis without our reconstruction is absolutely intractable with an $\mathcal{O}(2^n)$ algorithm. However, as we can see from Figure 13 even in case of failed reconstructions our analysis reduces the number of nodes by 30-40% in any optimization level, and may allow tractability in the average case, going from 2^{40} comparisons to less than 2^{30}.

Given the results, we can answer RQ1 as follows:

The average percentage of correctly reconstructed functions is around 90% for the O0 optimization level, 70% for Os and 45% for O2. Even in the case of failed reconstructions, the number of nodes is reduced by 35-45%, and enables tractability in the average case.

B. RQ2: CORRECTNESS

For the second research question, we want to analyze how accurate is the detection rate of cloned functions in our approach, while also estimating the change in accuracy varying the approach parameters. Specifically, we first analyze the clone detection using structural analysis only, then semantic analysis only and, finally, a combination of the two. We close this analysis by comparing our tool with two existing state-of-the-art products: BinDiff [17] and DeepBinDiff [39].

For this Research Question we compiled GNU coreutils[3] tagged `v.9.31` on Linux for the `x86_64` and `aarch64` architectures with optimization level `O2`. We chose *coreutils* for the reason that, being composed by several programs in a single codebase, we expect to find more clones than by choosing two completely random binaries. In the "same architecture" analysis we ran our tool comparing all functions belonging to the 108 *coreutils* binaries built for `x86_64`, reporting both intra-project and inter-project clones. Similarly for the "cross architecture" analysis we added also the functions belonging to the same binaries built for `aarch64`. As highlighted in Table 1, this time we did not strip the binaries: in fact, using non-stripped binaries allows us to investigate false negatives by matching the various function names. Note that our approach supports also sub-function granularity, as presented in Section III-E, however, to reduce the amount of manual analysis to be done, we limit the study to function granularity.

Unlike most tools in literature [17], [18], [31], ours compares not only pairs, but entire sets of similar functions. For this reason, the metrics were defined as follows: for each reported clone class A, the most similar clone class B was retrieved from the ground truth using Jaccard Similarity. With

[3]https://www.gnu.org/software/coreutils/

TABLE 2. True positives, False Positives, False Negatives, Precision and Recall in clone detection using structural analysis alone, varying the minimum tree depth threshold θ. Results obtained comparing all the coreutils binaries together.

	Same architecture (x86_64)					Cross architecture (x86_64, aarch64)				
θ	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall
2	3337	3774	48	0.4693	0.9858	4826	8966	2406	0.3499	0.6673
3	2150	581	309	0.7873	0.8743	3733	2313	2101	0.6174	0.6399
4	1540	291	162	0.8411	0.9048	2888	1213	1452	0.7042	0.6654
5	687	68	129	0.9099	0.8419	1167	579	833	0.6684	0.5835
6	375	34	91	0.9169	0.8047	663	293	576	0.6935	0.5351

TABLE 3. True positives, False Positives, False Negatives, Precision and Recall in clone detection using semantic analysis alone, varying the minimum cosine similarity threshold. Results obtained comparing all the coreutils binaries together.

	Same architecture (x86_64)					Cross architecture (x86_64, aarch64)				
Min.Cosine Sim.	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall
0.95	93287	252978	7360	0.2694	0.9269	218512	867087	209742	0.2013	0.5102
0.98	12256	14320	2090	0.4612	0.8543	39171	41092	41092	0.3710	0.4880
0.99	7928	2648	307	0.7496	0.9627	16626	11555	17958	0.5900	0.4807
0.999	8922	1163	3724	0.8847	0.7055	17406	3701	28343	0.8247	0.3805

these two sets, True Positives (TP) are defined as $\#(A \cap B)$, False Positives (FP) as $\#(A \setminus B)$ and False Negatives (FN) as $\#(B \setminus A)$. Additionally, we added to the False Negatives count the elements of each unmatched clone class in the ground truth. The ground truth set was built with a combination of similar function name matching, existing state-of-the-art tools and manual analysis.

Table 2 shows the results using Structural Analysis only. When operating within the same architecture, for sufficiently complex functions (higher θ), this approach is capable of reaching a precision up to 91%. In this type of analysis, the False Positives are exclusively determined by functions with the same structure but different opcodes. For example, in the binary `pr`, the functions `hold_file` and `tzfree` are reported as clones despite being composed of completely different opcodes. This happens because their structure is essentially the same. Naturally, with a low threshold these false positives increase in number, but even with a low value as $\theta = 3$ we are capable of reaching a fairly high precision of almost 80%.

The same cannot be said for the cross architectural Structural Analysis. In this case the precision never goes above 70%. The main problem we identified by analyzing the results is the fact that our approach can correctly report big clone classes in their respective architectures but these classes have slightly different structures, mostly sequences with a different number of basic blocks, that result in different structural hashes. For this reason these clone classes are not merged into a single cross-architectural clone class. Naturally, we report these clones as False Positives/Negatives: despite being correctly identified as clones we are interested in cross-architectural clones only.

Table 3, instead, shows the results using Semantic Analysis only. Unlike similar approaches in source code clone detection [48], the minimum similarity threshold for our frequency vector is much higher. Even in CISC architectures such as `x86_64` that potentially can use thousands of opcodes, in practice most functions use a small subset of common opcodes. This problem is exacerbated in the cross architectural detection by our ‘‘opcode family’’ described in Section III-F that further reduces variance between functions, resulting in 80% precision with functions that have a similarity higher than 0.999. In addition to this problem, the semantic analysis has complexity $\mathcal{O}(n^2)$ in the number of functions, compared to the $\mathcal{O}(n)$ of the structural analysis, limiting its scalability.

TABLE 4. Precision and Recall in clone detection within the same architecture, using structural analysis and semantic analysis combined, varying their input thresholds.

	Same architecture (x86_64)		
	Min. Cosine Sim.		
θ	0.98	0.99	0.999
2	0.8494 / 0.9746	0.8702 / 0.9708	0.8752 / 0.7690
3	0.9148 / 0.9114	0.9280 / 0.9084	0.9306 / 0.9235
4	0.9178 / 0.9143	0.9345 / 0.9145	0.9363 / 0.9039
5	0.9627 / 0.8593	0.9876 / 0.8646	0.9901 / 0.7964
6	0.9658 / 0.8905	0.9888 / 0.7599	0.9907 / 0.8004

While discussing Table 2 we explained how most of the False Positives in the structural analysis are determined by clones with the same structure but different opcodes. For this reason, by using the results of the structural analysis and applying semantic analysis on them, we overcome this problem and obtain the results shown in Table 4. The Table reports only precision and recall, but we can clearly see how dramatic the improvement in precision is: even for simpler functions composed of three nested structures both precision and recall are now above 91%. For complex functions the precision can reach up to 99% with our best result being 474 True Positives and only 4 False Positives in one configuration. The remaining false positives are due to the granularity of our approach returning block-level clones but being treated as function-level clones by our experimental settings, in particular for $\theta = 2$.

TABLE 5. Precision and Recall in clone detection across different architectures, using structural analysis and semantic analysis combined, varying their input thresholds.

	Cross architecture (x86_64, aarch64)		
	Min. Cosine Sim.		
θ	0.98	0.99	0.999
2	0.7311 / 0.5071	0.7334 / 0.5087	0.7516 / 0.4645
3	0.7241 / 0.5402	0.7368 / 0.5476	0.7449 / 0.5250
4	0.7691 / 0.5504	0.7679 / 0.5647	0.7720 / 0.5402
5	0.7204 / 0.5398	0.7240 / 0.5145	0.7320 / 0.5263
6	0.7410 / 0.4922	0.7430 / 0.4908	0.7500 / 0.4888

This solution, however, does not work for cross architectural clones, as can be seen in Table 5. Here the main problem lies in the structural analysis not correctly mixing the various clone classes and this problem can not be fixed by the semantic analysis.

TABLE 6. Time required to perform the structural analysis, semantic analysis and combined analysis in both same architecture and cross architecture using $\theta = 3$ and min similarity of 0.99. Results obtained analyzing all coreutils binaries together.

Analysis	Time same arch.(μs)	Time cross arch.(μs)
Structural only	5066	8922
Semantic only	16111921	46399417
Combined	543672	497643

Finally, Table 6 shows the time required for each approach. We can clearly see how the semantic analysis is 1000 times slower than the structural one, while the combination of the two still manages to reach an acceptable time, due to the initial filtering performed by the structural analysis.

TABLE 7. Precision and number of detected clones in pairwise function detection of several state-of-the-art approaches. The listed programs have been compared against `du`.

bin	BinCC (ours)	BinDiff	DeepBinDiff
dir	0.9593 (172)	0.9333 (105)	0.9368 (95)
ls	0.9593 (172)	0.9333 (105)	0.9167 (96)
mv	0.9704 (169)	0.9739 (115)	0.9245 (106)
cp	0.9652 (144)	0.9783 (92)	0.8295 (88)
sort	0.9923 (131)	0.9670 (91)	0.9157 (83)
du	0.9574 (188)	0.9937 (159)	0.9933 (150)
csplit	0.9574 (94)	0.9254 (67)	0.9194 (62)
expr	0.9489 (98)	0.9677 (62)	0.9062 (64)
nl	0.9444 (90)	0.9672 (61)	0.9828 (58)
ptx	0.9266 (109)	0.9444 (72)	0.9254 (67)
split	0.9375 (96)	0.9538 (65)	0.9831 (59)
mean	0.9562	0.9580	0.9303

In Table 7 we compare our approach against BinDiff [17], a commercial tool for binary diffing, and DeepBinDiff [39], a research tool performing the same task using deep learning. Given that both BinDiff and DeepBinDiff support only pairwise comparison, and none of them support cross-architectural comparison, we compared eleven binaries in the coreutils package of different size against `du`. In particular we used our tool combining structural and semantic analysis with a low threshold in order to match as many clones as possible ($\theta = 2$). For BinDiff and DeepBinDiff, instead, we report as clone a pair of functions having at least half their basic blocks matching. Using higher threshold for all tools would result in perfect precision, but would miss most function pairs.

We can see that BinCC, our tool, even with low thresholds, can emit more clones compared to the state-of-the-art. This is due to our tool being capable of emitting intra-project clones that, depending on the use case, may be desirable or not. The precision of the detected clones is similar with respect to the other state-of-the-art tools. We determined that our tool, similarly to DeepBinDiff, fails in cases where functions have identical implementation modulo a different function call. For example, they both detect `xcalloc` and `xmalloc` as clones, being these two functions different only in the allocation function used. The results obtained from DeepBinDiff are comparable with the results presented in its original paper [39], while for BinDiff we obtained slightly more accurate results compared to previous experiments [31].

A huge difference instead, can be noted in the speed. Table 8 shows a comparison between our tool and DeepBinDiff. BinDiff using Ghidra requires to manually disassemble the file, and for this reason it is not listed. The Table clearly shows that our tool is order of magnitudes faster than the competition, in some cases reaching difference of a factor 10^3 (e.g. the comparison `du-du`). In addition, DeepBinDiff can compare only two binaries together, requiring to perform the analysis for any combination of executables, whereas our tool can report the clones for all 108 coreutils binaries together in less than a minute.

For RQ2 we can thus conclude:

Even without using semantic analysis our approach is capable of reaching more than 90% precision if the function is complex enough. For simpler functions, this precision can be reached by combining the structural analysis with a semantic one, sacrificing some detection speed. Nonetheless, our tool is still order of magnitude faster than the competition and can achieve similar precision.

C. RQ3: USE-CASE

After performing the evaluation on a controlled case with unstripped binaries in our possession, in this section we want to simulate a use-case by checking the detection rate on stripped-only, real-case binaries.

Given that after the strip phase every information about the function name is lost, building a ground truth set requires manually checking every function. For this reason, we adopted an approach similar to the one of Hemel et al. [24]: we compiled several binaries with the default options and static linking. After that, we ran the comparison between the binary itself and the libraries reported to be used at link time. Additionally, we performed the check for a limited set of extra libraries unrelated to the project. In this case, being more of a use-case, we wanted to check more consistent functions and thus used higher thresholds. In the dataset summary of Table 1, we reported the optimization for this Research Question as "unknown": being this a real-case scenario, we used binaries without knowing the original compilation options. We used a value θ of 7 and a minimum number of nodes of 7 for the reconstruction and 15 for the CFG.

TABLE 8. Time required to compare `du` with the binary listed in the column bin, in seconds. The size column contains the combined size of `du` and the target binary. Times have been counted from program invocation until program termination.

bin	size (KiB)	BinCC (ours)	DeepBinDiff
dir	1081	3.40s	1409s
ls	1081	3.18s	1432s
mv	1045	3.43s	1999s
cp	978	3.22s	1740s
sort	952	2.90s	1216s
du	921	2.80s	2043s
csplit	682	2.55s	659s
expr	673	2.44s	652s
nl	642	2.43s	527s
ptx	749	2.58s	771s
split	701	2.43s	702s

Table 9 shows the results for *busybox* tag 1_31_0 which uses *libm* and *libresolv*. Our tool correctly reported function reuse only in the libraries statically linked with the binary. We then ran the same analysis against those libraries compiled for `aarch64` and obtained similar results except for *libm* that was not detected: of the 40 functions checked in `x86_64`, 0 were analyzed in `aarch64`. The reason for this is that multiple functions were under the threshold and thus not analyzed, probably due to the smaller CFG in the second architecture. The number of analyzed functions for the other libraries were similar instead.

However, in order to not bias the evaluation given that we purposely used libraries unrelated to the project, we also reran the evaluation with 800 libraries usually present in an Ubuntu Linux install. In this specific case, we found 407 libraries that had function reuse and 393 that did not. We did not check if all the reports were false positives or negatives, because this would require a prohibitively high amount of time having to manually understand and analyze over 8000 functions in assembly. However, we found that most of the reported libraries are dependent upon *libc*, that is statically linked with the executable we analyzed.

For this reason, we plan to conduct a more controlled test, analyzing also the dependencies between the various libraries.

We can then answer RQ3 as follows:

By analyzing a stripped real-case binary executable our approach detects the libraries statically linked with it and also the dependencies of those libraries. Libraries unrelated with them are not detected

TABLE 9. Results of the library and function usage detection for busybox. The column "used" refers to functions in the library that were used inside busybox. "checked" refers to the amount of functions checked from that library.

library name	used	checked
libresolv.so.2	1	2
libm.so.6	2	40
libjpeg.so.8.1.2	0	8
libtiff.so.5.3.0	0	7
libpng16.so.16.34.0	0	8
libMagick++-6.Q16.so.7.0.0	0	4
libFLAC.so.8.3.0	0	1
libsamba-util.so.0.0.1	0	21
libXext.so.6.4.0	0	20

D. RQ4: PERFORMANCE

The last research question evaluates the performance and the scalability of our analysis. In order to do so, we used a subset of the same dataset used for RQ1, limited to `x86_64` and `O2`. These times were recorded on a machine mounting an Intel Xeon E5-2620 @ 24x 2.5GHz with 64GB of RAM, limited to a single core and with hyper-threading disabled.

Figure 14 shows the time required for the disassembly of the executables. We can observe that the time scales linearly with respect to the executable size, and it is in the order of hundredth of seconds.

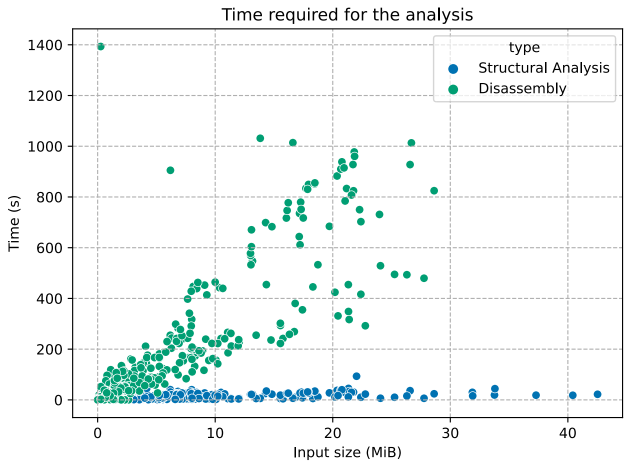


FIGURE 14. Time required for disassembly and reconstruction step on x86_64 O2.

However, by observing Figure 15, showing the time required for the structural analysis only, we can note that the time required is in the order of tenth of seconds. The entire procedure is thus completely dominated by the time required by the disassembly on which we depend, rather than the time required by our analysis. This can be seen clearly also in Figure 16 showing the composition of time required to perform the combined analysis. In the Figure we can note how the disassembly operation takes half the total time of the entire analysis, represented by the line. Moreover we can note that our approach working with a 158MiB executable requires the same time as existing approaches on a 400KiB, as shown in Table 8.

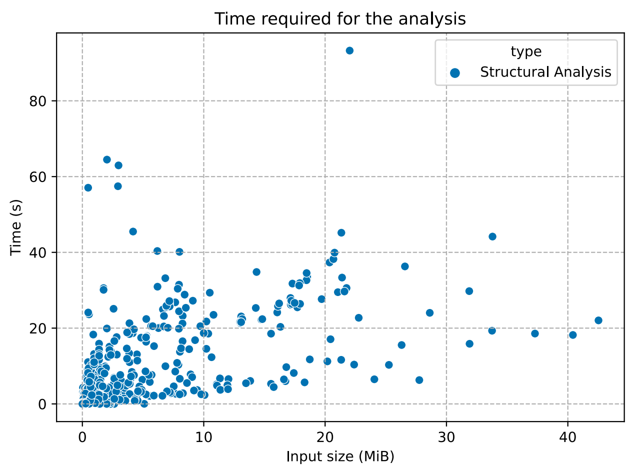


FIGURE 15. Time required for the CFG reconstruction step.

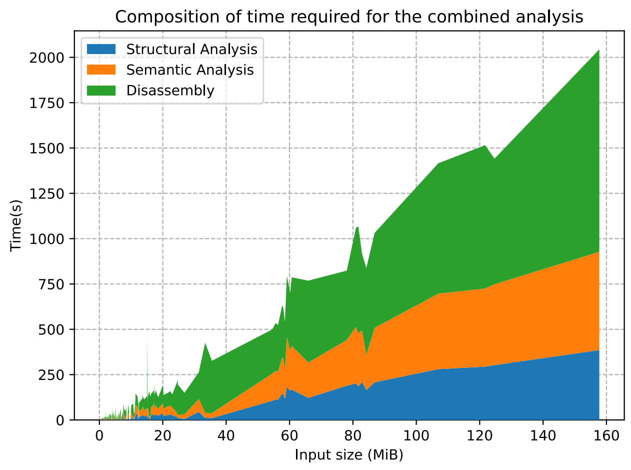


FIGURE 16. Time required for the combined analysis. The line represents the total amount of time required, while the colors represent the portion of time taken by each analysis step.

We experimented with the disassembler, and managed to perform a faster disassembly by analyzing only the function calls instead of performing a full binary analysis. However, the reconstruction accuracy suffered greatly: the fast disassembly took around the same time as our structural analysis, but the reconstruction accuracy presented in RQ1 dropped by a flat 20% in every optimization level. For this reason, we decided not to report this analysis with the fast disassembly option.

For RQ4 we can thus conclude:

Our analysis scales linearly and is completely dominated by the time required to perform the disassembly.

V. LIMITATIONS AND THREATS TO VALIDITY

A. PREDICATION

In some architectures, such as ARM, predication is used as an alternative to branching by converting flow dependency to data dependency. Predication works by associating instructions with a *predicate*, a boolean value, usually a CPU flag, used to indicate if the instruction is allowed to execute. If this boolean value does not hold at the time the instruction should be executed, that instruction does not modify the architectural state, with the advantage of not breaking the CPU pipeline. As an example, consider the following code:

```
0x410 cmp r0, 0
0x414 cmpne r1, 0
0x41C movne r0, 5
0x420 moveq r0, 6
0x424 bx lr
```

The first instruction executes a comparison and sets the CPU flags according to the result. If the flags are set as true, the second and third instructions do nothing, and the fourth is executed, otherwise the second is executed which changes the flags values again for when the third will be executed. The problem with predication thus lies in the fact that a CFG reconstructed just by looking at jumps will miss the flow hidden beneath these instructions. In our implementation we decided to ignore predication given that only two mnemonics, namely CMOV and SETcc, supports these operations in x86_64, while in aarch64 these are mostly deprecated given the recent advancements with branch predictors [49].

B. MISMATCHED COMPILER CONFIGURATIONS

In addition to the results showed in Section IV-B, we briefly analyzed potential clones between binaries compiled with different compilers or optimization settings.

Results show that our algorithm fails at finding potential clones when the compiler or the optimization flags are different: these results are due to the fact that code is rearranged during the data flow analysis performed by the compiler, and thus the output dramatically changes depending on the analyses performed at compilation time.

This was originally highlighted by Sæbjørnsen et al. [22] and we can confirm that not only the compiled binary is semantically different at various optimization levels but also structurally different, and we expect every CFG-based approach to suffer from this limitation. In order to overcome this problem, however, a previous work of us was focused on determining automatically the compiler and optimization level used in a binary, in order to mitigate failed analyses due to mismatched compiler configurations [50].

C. DISASSEMBLER DEPENDENCY

In this work, the starting point of our analysis is the disassembler. We relied on an external tool as there is no novelty in implementing our own and deemed the process as out-of-scope for this work. However, this means the goodness of our structural analysis is dependent on the disassembly quality: using a different disassembler may result in a different qualitative result in terms of structural analysis ability and clone detection precision. We highlighted this fact in Section IV-D by citing the fast analysis: using a faster but less accurate disassembly resulted in a flat drop of 20% accuracy in the structural analysis alone.

VI. CONCLUSION

In this paper, we presented our approach for finding function reuse and clones using a high-level refined CFG. We implemented this analysis and conducted several tests in 24 million functions in the `x86_64` and `aarch64` architectures.

Results show that our analysis is faster than existing approaches, and the time required is dominated by the disassembly step on which it depends. Our technique can be applied without any training step in 20 different architectures, analyzing any given number of executables at the same time.

When performing the comparison, our work showed to be capable of detecting function clones with precision ranging from 91% to 99% when comparing binaries from the same architecture and 75% when comparing binaries in different architectures. In addition, we showed that our approach can detect the presence of functions coming from a particular library.

VII. REPLICATION

The dataset used in our study can be found on Zenodo at the following URL [47], while source code can be found publicly on GitHub.[4] On GitHub, the branch `experiments` contains all the experimental data and results used in this paper.

REFERENCES

[1] S. Morasca, "Why do developers adopt open source software? Past, present and future," in *Proc. IFIP Int. Conf. Open Source Syst.* Montreal, QC, Canada: Springer, 2019, pp. 104–115.

[2] F. Fischer, K. Bottinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack overflow considered harmful? The impact of copy & paste on Android application security," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 121–136.

[3] C. Ragkhitwetsagul, J. Krinke, M. Paixao, G. Bianco, and R. Oliveto, "Toxic code snippets on stack overflow," *IEEE Trans. Softw. Eng.*, vol. 47, no. 3, pp. 560–581, Mar. 2021.

[4] D. M. German, M. Di Penta, Y.-G. Gueheneuc, and G. Antoniol, "Code siblings: Technical and legal implications of copying code between applications," in *Proc. 6th IEEE Int. Work. Conf. Mining Softw. Repositories*, May 2009, pp. 81–90.

[5] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *J. Univers. Comput. Sci.*, vol. 8, no. 11, p. 1016, 2002.

[6] H. Zhang and K. Sakurai, "A survey of software clone detection from security perspective," *IEEE Access*, vol. 9, pp. 48157–48173, 2021.

[7] N. Yoshida and E. Choi, "Clone evolution and management," in *Code Clone Analysis*. Berlin, Germany: Springer, 2021, pp. 197–208.

[8] K. Inoue and C. K. Roy, *Code Clone Analysis*. Berlin, Germany: Springer, 2021.

[9] D. Yu, J. Wang, Q. Wu, J. Yang, J. Wang, W. Yang, and W. Yan, "Detecting Java code clones with multi-granularities based on bytecode," in *Proc. IEEE 41st Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, Jul. 2017, pp. 317–326.

[10] I. Keivanloo, C. K. Roy, and J. Rilling, "Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning," in *Proc. 6th Int. Workshop Softw. Clones (IWSC)*, Jun. 2012, pp. 36–42.

[11] P. M. Caldeira, K. Sakamoto, H. Washizaki, Y. Fukazawa, and T. Shimada, "Improving syntactical clone detection methods through the use of an intermediate representation," in *Proc. IEEE 14th Int. Workshop Softw. Clones (IWSC)*, Feb. 2020, pp. 8–14.

[12] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, "Value-based program characterization and its application to software plagiarism detection," in *Proc. 33rd Int. Conf. Softw. Eng.*, May 2011, pp. 756–765.

[13] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu, "A first step towards algorithm plagiarism detection," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2012, pp. 111–121.

[14] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "DiscovRE: Efficient cross-architecture identification of bugs in binary code," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 58–79.

[15] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 709–724.

[16] C. Ragkhitwetsagul and J. Krinke, "Using compilation/decompilation to enhance clone detection," in *Proc. IEEE 11th Int. Workshop Softw. Clones (IWSC)*, Feb. 2017, pp. 1–7.

[17] H. Flake, "Structural comparison of executable objects," in *Proc. Int. GI Workshop Detection Intrusions Malware Vulnerability Assessment*, 2004, pp. 161–174.

[18] J. Oh, "Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries," in *Proc. Blackhat Tech. Secur. Conf.*, 2009, pp. 1–28.

[19] M. Sharir, "Structural analysis: A new approach to flow analysis in optimizing compilers," *Comput. Lang.*, vol. 5, nos. 3–4, pp. 141–153, Jan. 1980.

[20] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, Jan. 1976.

[21] W. Amme, T. S. Heinze, and A. Schafer, "You look so different: Finding structural clones and subclones in Java source code," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2021, pp. 70–80.

[22] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proc. 18th Int. Symp. Softw. Test. Anal. (ISSTA)*, 2009, pp. 117–128.

[23] A. Schulman, "Finding binary clones with opstrings function digests: Part III," *Doctor Dobbs J.*, vol. 30, no. 9, p. 64, 2005.

[24] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proc. 8th Work. Conf. Mining Softw. Repositories (MSR)*, 2011, pp. 63–72.

[25] F. Engel, R. Leupers, G. Ascheid, M. Ferger, and M. Beemster, "Enhanced structural analysis for C code reconstruction from IR code," in *Proc. 14th Int. Workshop Softw. Compil. Embedded Syst. (SCOPES)*, 2011, pp. 21–27.

[26] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Proc. 22nd USENIX Secur. Symp.*, 2013, pp. 353–368.

[27] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[28] M. Bourquin, A. King, and E. Robbins, "BinSlayer: Accurate comparison of binary executables," in *Proc. 2nd ACM SIGPLAN Program Protection Reverse Eng. Workshop (PPREW)*, 2013, pp. 1–10.

[29] D. Gao, M. K. Reiter, and D. Song, "BinHunt: Automatically finding semantic differences in binary programs," in *Proc. Int. Conf. Inf. Commun. Secur.* Cham, Switzerland: Springer, 2008, pp. 238–255.

[30] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice,'" in *Proc. 2nd ACM SIGPLAN Program Protection Reverse Eng. Workshop (PPREW)*, 2013, pp. 1–6.

[31] J. Ming, D. Xu, Y. Jiang, and D. Wu, "BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 253–270.

[32] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "BinGo: Cross-architecture cross-OS binary search," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2016, pp. 678–689.

[33] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proc. IEEE/ACM 25th Int. Conf. Program Comprehension (ICPC)*, May 2017, pp. 88–98.

[34] Y. Hu, Y. Zhang, J. Li, H. Wang, B. Li, and D. Gu, "BinMatch: A semantics-based hybrid approach on binary code clone analysis," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2018, pp. 104–114.

[4]http://github.com/davidepi/bincc

[35] Z. Ma, H. Ge, Y. Liu, M. Zhao, and J. Ma, "A combination method for Android malware detection based on control flow graphs and machine learning algorithms," *IEEE Access*, vol. 7, pp. 21235–21245, 2019.
[36] H. Xue, G. Venkataramani, and T. Lan, "Clone-slicer: Detecting domain specific binary code clones through program slicing," in *Proc. Workshop Forming Ecosystem Around Softw. Transformation (FEAST)*, 2018, pp. 27–33.
[37] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 480–491.
[38] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," 2018, *arXiv:1808.04706*.
[39] Y. Duan, X. Li, J. Wang, and H. Yin, "DeepBinDiff: Learning program-wide code representations for binary diffing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–16.
[40] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 472–489.
[41] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 363–376.
[42] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, 2009.
[43] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. 23rd Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2007, pp. 421–430.
[44] A. Gonzalvez and R. Lashermes, "A case against indirect jumps for secure programs," in *Proc. 9th Workshop Softw. Secur., Protection, Reverse Eng. (SSPREW)*, 2019, pp. 1–10.
[45] C. Cifuentes and M. Van Emmerik, "Recovery of jump table case statements from binary code," *Sci. Comput. Program.*, vol. 40, nos. 2–3, pp. 171–188, Jul. 2001.
[46] G. Fowler and P. Vo. (May 1991). *Fowler/Noll/Vo (FNV) Hash*. [Online]. Available: http://isthe.com/chongo/tech/comp/fnv
[47] D. Pizzolotto and K. Inoue. (Apr. 2021). *Binary Software Compiled for Different Architectures With Different Optimization Levels*. [Online]. Available: https://doi.org/10.5281/zenodo.4659370
[48] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proc. 38th Int. Conf. Softw. Eng.*, May 2016, pp. 1157–1168.
[49] *ARMV8 Instruction Set Overview*, document PRD03-GENC-010197, Architecture Group, vol. 15, no. 11, Oct. 2011.
[50] D. Pizzolotto and K. Inoue, "Identifying compiler and optimization level in binary code from multiple architectures," *IEEE Access*, vol. 9, pp. 163461–163475, 2021.

DAVIDE PIZZOLOTTO received the B.S. and M.S. degrees in computer science from the University of Trento, in 2015 and 2019, respectively. He is currently pursuing the Ph.D. degree with the Graduate School of Information Science and Technology, Osaka University. Previously, he was a Research Assistant at Fondazione Bruno Kessler (FBK). His research interests include code obfuscation, binary code analysis, and source code analysis and transformation.

KATSURO INOUE received the Ph.D. degree from Osaka University, in 1984. He was an Associate Professor with the University of Hawaii at Manoa, from 1984 to 1986. After becoming an Assistant Professor at Osaka University, in 1986, he was a Professor, from 1995 to 2022. He is currently a Professor at Nanzan University. His research interests include software engineering, especially software maintenance, software reuse, empirical approach, program analysis, code clone detection, and software license/copyright analysis.