# Efficient Manipulation of Control Flow Models in Evolving Software

Tomáš Fiedor[1,2][0009−0009−2596−9399], Jiří Pavela ✉[1,2][0000−0002−2579−827X], Adam Rogalewicz[1][0000−0002−7911−0549], and Tomáš Vojnar[1,3][0000−0002−2746−8792]

[1] Brno University of Technology, Faculty of Information Technology, Czechia
{ifiedortom,ipavela,rogalew}@fit.vutbr.cz
[2] Red Hat Czech, Czechia
[3] Masaryk University, Faculty of Informatics, Czechia
vojnar@fi.muni.cz

**Keywords:** Evolving software · Static analysis · Dynamic analysis · Diff analysis · Call graph · Control flow graph · Function Matching.

**Abstract.** When looking for certain kinds of software bugs, successive versions of software are compared. Performance-related bugs are a notable example. Methods used for detecting such bugs are, however, expensive and need to be applied carefully. At the same time, current software development is rapid, with new software versions released everyday. In this paper, we aim at two particular ways how to optimize difference analyses of performance (but possibly other aspects of the software too) of successive software versions. Namely, we propose (1) an efficient layered representation of the program control flow spanning across the program history, and (2) methods for efficient matching of pairs of corresponding functions in different software versions and for selecting those whose differential analysis should be performed. We have implemented our approach and performed experiments on two selected versions of the CPython project. The results indicate that our approach is a promising direction for improving the performance analysis of real world programs.

## 1 Introduction

Over the last years, the software development community has seen a massive rise in the popularity of *Continuous Integration and Continuous Delivery (CI/CD)* practices. In particular, CI/CD aims to automate building, testing, and deploying software projects during their entire development lifecycle, naturally leading to *evolving software* that is constantly updated with new features without breaking the build.

As more developers adopted CI/CD in their software projects, numerous success stories have been publicly shared, thus further accelerating interest and demand for CI/CD solutions among software developers. Numerous companies reported impressive results ranging from reducing the development costs by 78% (HP), to increasing the number of deployments to more than ten per day (Flickr) or 50 million a year (Amazon) [9]. Moreover, an empirical study [8] on open-source projects found out that projects using CI/CD practices, as compared to projects without CI/CD, on average (1) save 1.6 hours on integrating pull requests; (2) manage to release twice as often; (3) catch bugs earlier; and (4) help the developers avoid breaking the build.

In their current state, most CI/CD test environments focus on finding functional bugs only, i.e., bugs that lead to software crashes and/or incorrect results. Other types of bugs, such as *performance bugs*, are often overlooked. However, a study of Mozilla Firefox [16] concluded that, compared to any other type of bugs, performance bugs (1) take more time to fix – up to 2.8 times more than even security fixes; (2) require more experienced developers to fix; and (3) the fixes are spanning up to 2.6 times more files. Moreover, if such bugs remain in the software unnoticed for a long time (thus becoming the so-called *dormant* bugs), the time needed to fix them almost doubles [5]. In extreme cases, this can even lead to hundred-million dollar projects being abandoned after several years of intense development [7].

Recently, there have been attempts to integrate performance testing to CI/CD [10,2]. However, these solutions usually provide only basic performance testing support (e.g., regression testing, micro benchmarking, or load testing) and focus on the latest software version only. In our previous work [6], we have introduced *Perun*: a novel tool-suite for performance testing and analysis. Perun supports more complex performance analyses that require precise profiling and can possibly take into account multiple previous versions of the software. Such larger perspective is generally necessary to find root causes of hard-to-detect performance issues, especially in the case of so-called *creeping degradations*: performance degradations of software caused slowly over the time.

In order to integrate more complex performance analyses, e.g., into a CI/CD pipeline, we need them to be efficient. A possible approach is to analyse only those source files that have been changed since the last analysis. This is successfully used in practice, e.g., by Facebook/Meta Infer [1] in the area of static analysis. In our ongoing work on efficient performance analyses, we build on this principle and focus our analysis to, among other, code fragments with control flow changes. However, this poses numerous challenges in terms of efficient representation, versioning, storage, and difference analysis of control flow models, i.e., control flow and call graphs, in evolving software.

In this work, we address the above challenges by proposing (1) an efficient *layered representation* of control flow models; (2) a hierarchical version labeling system – using, among others, hashes of *version control system (VCS)* versions and local repository changes – along with accompanying algorithms for storage, retrieval, and intelligent lazy deletion of individual versions; (3) six new heuristics for function matching adapted specifically for evolving software to improve the matching accuracy; and (4) a generic algorithm for difference analysis of call and control flow graphs that builds on the matching heuristics. We have implemented the proposed solutions in the particular context of the Perun analyser and present a use-case demonstration with experimental results that confirm these solutions can indeed bring significant efficiency and false match rate improvements. Moreover, we believe that the data structures and algorithms that our solution is based on are general enough to be useful in other tools that leverage control flow information, allowing them to be used efficiently in CI/CD.

## 2 Preliminaries

In Perun, we employ, among other, two kinds of program *control flow models (CFM)*: (1) Control Flow Graphs (CFGs) to represent the structure of particular functions, and,

(2) Call Graphs (CGs) to represent the calling relation between particular functions. Their usage is quite versatile: we use CFGs to, e.g., match functions that were renamed between two versions, allowing us to omit them from the subsequent analysis if just the name changed, and we can potentially use CGs to, e.g., improve the precision of the results reported to the user by supplying a more precise trace leading to a detected bottleneck.

CFGs consist of nodes and edges where nodes represent *basic blocks* – sequences of instructions without branching; and edges represent direct and indirect jumps between basic blocks.

**Definition 1 (Basic Block).** *A* basic block $BB$ *is a straight-line sequence of instructions* $i_0, i_1, \ldots, i_{n-1}$ *with no branches in (apart from the entry point) and no branches out (apart from the exit point). We denote by* $\mathrm{len}(BB)$ *the length of the basic block, i.e., the number of instructions in* $BB$.

In the rest of the paper, we assume dealing with *instructions* from the assembly language of the `x86_64` architecture, but the proposed algorithms should be applicable to any other CPU architecture or machine-level instruction set without a loss of generality. The concrete instruction set and architecture are relevant as they determine the set of registers and control flow related instructions, such as jumps or calls.

**Definition 2 (Program Control Flow Graph).** *The* control flow graph *(CFG) of a program* $P$ *is a directed graph* $CFG_P = (\mathcal{B}_P, \mathcal{C}_P, T_P)$ *where* $\mathcal{B}_P$ *is the (finite) set of basic blocks of* $P$*,* $\mathcal{C}_P \subseteq \mathcal{B}_P \times \mathcal{B}_P$ *is the control flow relation, and* $T_P : \mathcal{C}_P \to \{\mathrm{fallthrough}, \mathrm{jump}\}$ *is a mapping of edges to their type.*

Note that we extend the classical definition of a CFG with an additional mapping $T_P$ that assigns each edge its type. When a conditional branching occurs in a CFG (e.g., as a result of a comparison or a jump instruction at the end of some basic block), one or more of the outgoing edges usually represent the control flow to the jump destination ($\mathrm{jump}$ edges), while a single edge represents the control flow to the instruction directly succeeding the jump condition (the $\mathrm{fallthrough}$ edge), which is taken when the jump condition is not satisfied. This mapping has one practical advantage: it allows us to deterministically traverse two or more CFGs in a lockstep and compare their basic blocks and/or structure.

Manipulation with the CFG of the entire program is usually impractical as many analyses are performed on the function level. Even in the case of inter-procedural analyses (such as those we will introduce in Section 4), we often need to limit the control flow to the scope of individual functions.

**Definition 3 (Function Control Flow Graph).** *Let* $CFG_P$ *be the CFG of a program* $P$*. The CFG of a function* $f$ *of* $P$ *is the subgraph* $FCFG_f = (\mathcal{B}_f, \mathcal{C}_f, T_f)$ *of* $CFG_P$ *where* $\mathcal{B}_f \subseteq \mathcal{B}_P$ *is the finite set of basic blocks of the function* $f$*,* $\mathcal{C}_f \subseteq \mathcal{B}_f \times \mathcal{B}_f \subseteq \mathcal{C}_P$ *is the control flow relation of* $f$*, and* $T_f : \mathcal{C}_f \to \{\mathrm{fallthrough}, \mathrm{jump}\}$ *such that* $T_f \subseteq T_P$.

Call graphs consist of nodes that represent functions in a program and edges that represent the caller/callee relation between the functions.

**Definition 4 (Call Graph).** *The* call graph *(CG) of $P$ is a multi-rooted directed graph $CG_P = (\mathcal{F}_P, \mathcal{E}_P, \mathcal{R}_P)$ where $\mathcal{F}_P$ is the set of functions of $P$, $\mathcal{E}_P \subseteq \mathcal{F}_P \times \mathcal{F}_P$ is the call relation (relating callers and callees), and $\mathcal{R}_P \subseteq \mathcal{F}_P$ is the set of $CG_P$ roots.*

Note that contrary to some of the established CG definitions, we require the CG to support multiple root nodes as, in practice, many types of programs, e.g., (shared) libraries, may expose multiple valid entry point functions.

## 3 Control Flow Models Representation

In our experience, a typical workflow of dealing with control flow models in program analysis can be summarized into three steps: (1) (re)construct the models from source or binary files, dynamic run traces, logs, or any other available resources; (2) analyse the models according to the problem domain; and (3) discard the models. If a particular model is required again later on, e.g., as a part of some more complex aggregate analysis, it is simply reconstructed on-demand once more. Although this approach works for simple enough use cases, it is insufficient for scenarios where repeated reconstruction of control flow models or their post-processing is too expensive—as is the case with Perun.

Notably, obtaining precise CGs and CFGs using static analysis on binary executable files is notoriously difficult and expensive, mainly due to *function pointers* or *dynamic dispatch calls* found in many modern programming languages. Moreover, such control flow models are generally unsound and incomplete when constructed using static analysis on binary files only. Similarly, constructing a precise CG using just dynamic analysis is often infeasible in practice as the number of potential execution paths through a program rapidly grows with the size of the codebase. Such control flow models are, however, at least sound but generally still incomplete [14].

A naive solution to this problem might be to simply cache or persistently store all the constructed models so that they can be accessed at a later time. However, such a simple approach is impractical for numerous reasons, e.g., the space requirements for storage of models would be needlessly high as each static and dynamic model would be stored separately, and the lookup for a model tied to a concrete project version would be difficult without a proper versioning scheme. Also, combining the statically and dynamically constructed models into a single more precise model requires further, potentially expensive, post-processing step on each retrieval given those models are stored separately.

That is why we propose an efficient representation of a CG that composes both static and dynamic models obtained by various tools or run configurations, while allowing to manipulate with any individual model or a combination of models. Our representation is based on the definition of CGs in Definition 4.

First, we extend CGs with a set of so-called *layers*. Each node or edge of the CG must then belong to at least one layer but may belong to multiple or even all layers in the graph. (Alternatively, one can view this such that each node and edge is labelled by a set of layer identifiers.) The layers correspond to a single specific model or a combination of some simpler models. Each layer is associated with a tuple $(source, config)$ with $source$ specifying the type of the model, e.g., static or dynamic, and $config$ specifying the analysis tool, the configuration of the tool used, and/or the parameters the

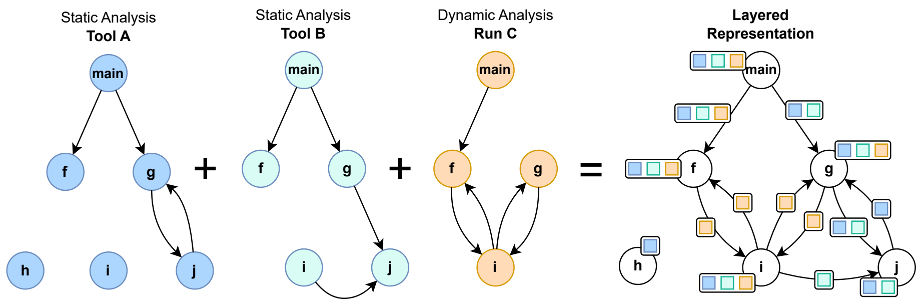
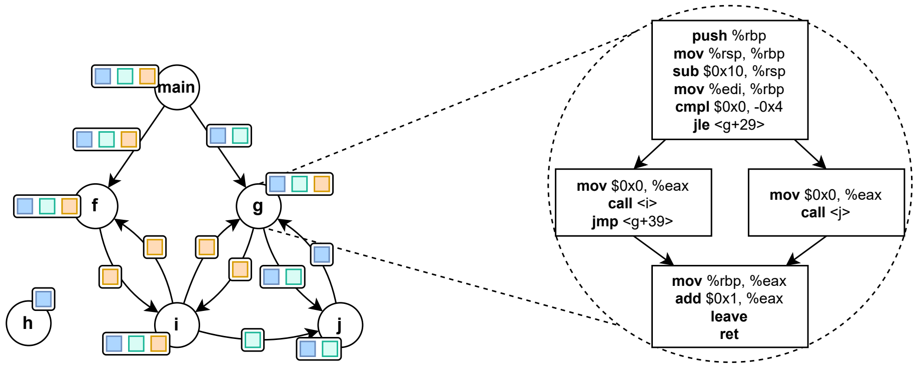


**Fig. 1.** An example of the proposed *layered* call graph representation with three distinct layers. Each color-coded layer corresponds to a concrete source of CG data, e.g., two different static analysis tools $A$ and $B$, and one dynamic run $C$. Together the layers form a single, potentially disconnected, graph structure.



```
push %rbp
mov %rsp, %rbp
sub $0x10, %rsp
mov %edi, %rbp
cmpl $0x0, -0x4
jle <g+29>
```

```
mov $0x0, %eax
call <i>
jmp <g+39>
```

```
mov $0x0, %eax
call <j>
```

```
mov %rbp, %eax
add $0x1, %eax
leave
ret
```

**Fig. 2.** A schematic illustration of the combined representation of a layered CG and a nested FCFG representation stored within the CG nodes.

given program was run or compiled with. Figure 1 shows an example of the proposed representation with three layers.

Second, the CG nodes maintain a link to the *function control flow graph (FCFG)* corresponding to the function represented by the node. Our implementation in Perun currently supports a single FCFG associated with a function only as in our experience there tends to be generally not so much variation between FCFGs obtained by different disassembly or analysis tools. However, extending the implementation to accommodate for multiple FCFGs or their combinations is straightforward. Figure 2 illustrates the inclusion of the FCFG representation within the CG nodes.

Note that Definition 4 defining CGs does not make any assumption about the connectivity of the graph (in the extreme case, the CG may contain no edges at all). It is critical to not impose any restrictions on the graph connectivity, at least in the representation itself, as disconnected CGs obtained by static analysis are quite commonly encountered in practice, e.g., as a result of callbacks registered within a third-party li-

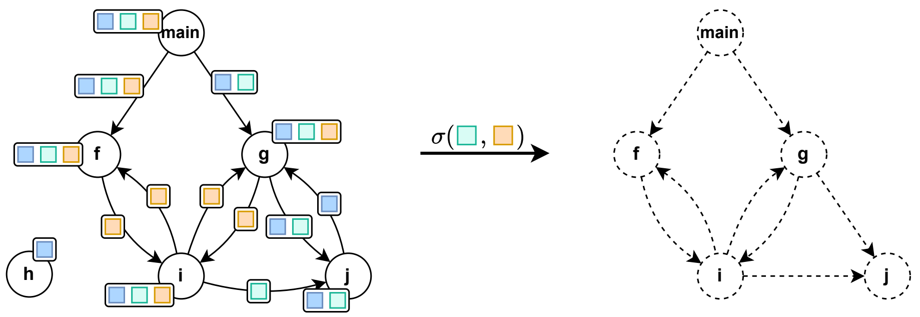


**Fig. 3.** An example of the proposed layer selection operator $\sigma$ used to obtain a combined view of two distinct layers $(\mathrm{static}, A)$ and $(\mathrm{dynamic}, C)$.

brary that cannot be properly analysed due to missing source code or debug information. A major benefit of this single-graph layered representation is that by extending the static CG with data from particular dynamic runs may cause previously disconnected parts of the CG to be reconnected as long as the dynamic runs visit the appropriate functions.

Using this representation and the accompanying features (such as the selection operator, versioning, or reuse of dynamic models introduced in the following sections), Perun is able to efficiently store even extensive control flow models combined from multiple sources, hence increasing the models precision. Perun can then leverage those models to quickly identify functions that have been changed in a new project version and focus on the performance of those particular functions, or apply other profiling optimizations that utilize call or control flow graph information.

## 3.1 CFM Layer Selection Operator

To easily manipulate individual layers or their combinations within the layered representation, Perun defines a *layer selection operator* $\sigma$ that allows individual analyses to choose a subset of layers that will form a *frozen view* over the selected layers. Our implementation of a frozen view does not allow direct updates to the view itself to avoid invalidation of iterators and possible inconsistencies, but provides access to the original CFM representation that can be updated. Note that the frozen view can be configured to preserve or exclude the links to the FCFG of individual CG nodes.

Additionally, we establish new auxiliary *convenience layers* to be used with the selection operator that predefine some commonly used combinations of layers, e.g., the *mixed* layer that combines all static and dynamic layers restricted to nodes and edges reachable from the root nodes.

Perun relies on the selection operator to obtain submodels of the entire control flow model restricted by custom user-defined conditions, e.g., a CFM view of concrete dynamic runs that may be reused in newer CFM versions as described in Section 4.5. Moreover, the matching and analysis algorithms introduced in Section 4 are designed to work on the view objects obtained by the selection operator. Figure 3 presents an example of the selection operator usage.

### 3.2 Call Graph Versioning

As projects evolve over time, new project versions are continuously released and both the CFG and CG may change as the project developers add new features or refactor the existing code. Some tools, such as Perun, need to frequently access some of those versions of the control flow models for comparison or other types of analyses. However, reconstructing precise multi-layered models can be extremely expensive, especially when using multiple static analysis tools and/or executing numerous dynamic runs to obtain control flow data. Therefore, we want to store the once constructed models for possible future use.

Perun already supports storing auxiliary data and models linked to concrete VCS versions, e.g., Git commits. However, versioning the models using the VCS versions only is generally insufficient as certain models might correspond to the so-called *dirty versions*[4], or even a different project setup or configuration. Naturally, we want to support the linkage of models to dirty versions or specific project configurations since we usually repeat the profiling when trying to fix a performance regression introduced in a concrete project version, hence introducing possibly many new dirty versions.

To solve the above issue with naive versioning, we propose a compound version identification $version(CFM) = (v, h, s, c, t)$ where $v$ is the VCS version (e.g., a commit hash); $h$ is a CFM version computed in a VCS-specific versioning algorithm (e.g., an SHA hash in Git) from files used for the CFM extraction; $s$ is the repository state, either *clean* or *dirty*; $c$ is a configuration name of the project setup, e.g., `release`, `debug`, `nightly-build`, etc.; and $t$ is the timestamp of the CFM creation.

As the number of stored *dirty models* (i.e., models linked to dirty VCS versions) and their size can become significant over time, especially for large actively maintained projects, we propose a lazy deletion algorithm to counter this issue. When traversing the project VCS history to retrieve a specific CFM version, the deletion algorithm identifies and deletes *obsolete* models, i.e., dirty models of the same project configuration in VCS versions other than the current `HEAD`[5].

## 4 Function Matching and Difference Analysis

Perun mainly focuses on detecting code changes between (recent) project versions. More precisely, Perun aims to automatically identify functions that were changed in a concrete (analysed) project version, as those functions are more likely to manifest new changes in performance as well as are more likely to be fixed by the developers [3,4,5]. Identifying changed functions naively by source code diff analysis leads to a significant number of false positives as many types of source code changes may have no impact on the behaviour or performance of a function, e.g., renaming variables, changing comments or refactoring code to macro expansions. Hence, the analysis needs

[4] A project version is considered *dirty* if it contains *uncommited* changes, i.e., changes that were not yet published in a commit.

[5] Using Git terminology, `HEAD` refers to the most current version in a given branch or a specific version if we find ourselves in a `HEAD`-detached state.

to be performed on a finer representation of functions, e.g., on the level of the control flow.

However, any such diff algorithm must first match functions from the new project version to their corresponding functions in the previous version[6]. We search for a bijection $m : \mathcal{F}'_{P_b} \leftrightarrow \mathcal{F}'_{P_t}$ where $\mathcal{F}'_{P_b} \subseteq \mathcal{F}_{P_b}$ and $\mathcal{F}'_{P_t} \subseteq \mathcal{F}_{P_t}$ are the largest possible subsets of functions from the previous (also called *baseline*) and current (called *target*) versions of a program $P$ for which such a bijection $m$ could be constructed w.r.t. some concrete chosen function correspondence criterion. This problem is computationally hard—indeed, even finding an isomorphism between two call graphs is a well-known NP-complete problem [11]. As such, it is in general infeasible to find a perfect solution to this problem. However, we believe we can find a good enough solution applicable to wide range of projects.

Our solution to this problem is inspired by existing works in this area [13,11] that are, however, mostly tailored for malware similarity analysis and other security-oriented analyses, and as such cannot rely on the availability of function names in the analysed programs. Hence, we propose a set of three matching heuristics based on *function summaries* inspired by [11], and three matching heuristics based on *function call context* that complement the function summary heuristics with a context-aware approach.

## 4.1 Function Matching Using Function Summaries

Function summaries are key characteristics about a function derived from its control flow data that can be exploited for fast identification of matching candidates. Formally, we define function summaries as follows.

**Definition 5 (Function Summary).** *Let $CFG_f$ be the function control flow graph of a function $f$. A* function summary *$S$ of $f$ is a tuple $S_f = (\beta, \phi, \epsilon, \iota, \mu)$ where $\beta$ is $|\mathcal{B}_f|$, i.e., the number of basic blocks in $f$; $\phi$ is the number of function call instructions in $CFG_f$; $\epsilon$ is $|\mathcal{C}_f|$, i.e., the number of edges in $CFG_f$; $\iota$ is $\sum_{b \in \mathcal{B}_f} len(b)$, i.e., the total number of instructions in $CFG_f$; and $\mu$ is $max_{b \in \mathcal{B}_f} len(b)$, i.e., the maximum length of a basic block in $CFG_f$. Let $\mathcal{S}_f$ denote the* summary set *of functions that share the same summary as $f$, including $f$.*

However, in our experience, using such summaries for function matching in real-world programs with hundreds of thousands of lines of code, such as CPython[7], can lead to a substantial number of false matching candidates. Large codebases typically contain many tiny functions, possibly consisting of only a single basic block. Consequently these blocks will most likely be matched as identical in terms of their function summaries. To combat this issue, we extended the function summaries with additional metrics to reduce the number of false matches. These extended summaries are then used only in the strictest matching heuristic.

[6] Informally, for each function we want to find its origin: a possibly renamed or changed function that is still enough structurally similar to the origin so one could argue that this function is indeed the evolved version of the origin. Whether two function implementations are enough structurally similar is of course highly subjective—one can hardly come up with an exact formal definition in this case.

[7] The reference C implementation of Python interpreter: https://github.com/python/cpython.

**Definition 6 (Extended Function Summary).** *Let $CFG_f$ be the function control flow graph of a function $f$. An* extended function summary *$ES$ of a function $f$ is a tuple $ES_f = (\nu, \beta, \phi, \epsilon, \iota, \mu)$ where $\nu$ is the name of the function $f$ and the remaining members of the tuple have the same meaning as in $S_f$.*

*Extended Function Summary Exact Matches.* Our first heuristic matches functions from $\mathcal{F}_{P_b}$ and $\mathcal{F}_{P_t}$ that have the same extended summaries. In particular, it matches functions that have identical name in the baseline and target versions, as well as the key characteristics of their FCFG, i.e., the function summaries. The goal is to quickly identify functions that have either not changed at all or changed just enough to not affect their FCFG in the target version.

*Function Summary Unique Renames.* Our second heuristic attempts to match functions that appear to be simply renamed without any substantial change that would alter their FCFG. First, we identify functions whose name appears exclusively in the target or baseline version denoted as $\mathcal{F}_{missing} \subseteq \mathcal{F}_{P_b}$ and $\mathcal{F}_{new} \subseteq \mathcal{F}_{P_t}$, respectively. Next, we match only those functions $f_m \in \mathcal{F}_{missing}$ and $f_n \in \mathcal{F}_{new}$ that have the same unique function summary, meaning no other functions $f'_m \in \mathcal{F}_{missing}, f'_m \neq f_m$ or $f'_n \in \mathcal{F}_{new}, f'_n \neq f_n$ belong to $\mathcal{S}_{(f_m, f_n)}$, i.e., share the same function summary.

*Function Summary Exclusive Renames.* The last heuristics heuristic aims to match renamed functions that did not meet the summary uniqueness criteria in the *Function Summary Unique Renames* heuristic and instead satisfy the summary and FCFG *exclusiveness* condition. We consider two functions $f_m \in \mathcal{F}_{missing}$ and $f_n \in \mathcal{F}_{new}$ to have an exclusive summary and FCFG[8] iff their summaries and FCFGs are equal, hence must belong to the same summary set $\mathcal{S}_{(f_m, f_n)}$, and $\nexists f' \in \mathcal{S}_{(f_m, f_n)} \cap (\mathcal{F}_{missing} \cup \mathcal{F}_{new})$ whose $FCFG_{f'}$ is equal to $FCFG_{f_m}$ or $FCFG_{f_n}$. The exclusiveness check is iteratively repeated until no new renames are found as the FCFG equality check result depends on the so-far known renames (see Section 4.4). Intuitively, this heuristic relaxes the requirement of summary uniqueness and instead, as a trade-off, introduces a new requirement of FCFG equality. Note that summary equality generally does not imply FCFG equality due to possible changes on the instruction level of individual basic blocks.

## 4.2 Function Matching Using Call Contexts

The next three heuristics are based on the notion of function's immediate neighbourhood in terms of the call relation.

**Definition 7 (Function Call Context).** *The* caller context *$\overline{\mathcal{C}}_f$ of a function $f \in P$ is the set of functions in $P$ that may call $f$, formally: $\overline{\mathcal{C}}_f = \{\overline{c} \mid (\overline{c}, f) \in \mathcal{E}_P\} \subseteq \mathcal{F}_P$. The* callee context *$\underline{\mathcal{C}}_f$ of $f$ is the set of functions in $P$ that may be called from $f$, formally: $\underline{\mathcal{C}}_f = \{\underline{c} \mid (f, \underline{c}) \in \mathcal{E}_P\} \subseteq \mathcal{F}_P$. The* function call context *$\overline{\underline{\mathcal{C}}}$ of $f$ is then $\overline{\underline{\mathcal{C}}}_f = \overline{\mathcal{C}}_f \cup \underline{\mathcal{C}}_f$.*

[8] Note that the FCFG equality check is done using the algorithm introduced in Section 4.4.

Note that in practice when comparing the $\overline{\mathcal{C}}$, $\underline{\mathcal{C}}$ or the entire $\overline{\underline{\mathcal{C}}}$ of two functions, the function names within the sets must be translated according to the already known function renames.

*Unique Function Call Contexts.* The first heuristic matches a function $f_b \in \mathcal{F}_{P_b}$ with $f_t \in \mathcal{F}_{P_t}$ iff they have the the same *unique* $\overline{\mathcal{C}}$, $\underline{\mathcal{C}}$ and their names are either identical or are candidates for a rename, i.e., $f_b \in \mathcal{F}_{missing} \subseteq \mathcal{F}_{P_b}$ and $f_t \in \mathcal{F}_{new} \subseteq \mathcal{F}_{P_t}$. Intuitively, this approach can match functions that might have different summaries and/or FCFGs as long as the functions $f_b$ and $f_t$ are used in exactly the same call contexts, thus nicely complementing the summaries heuristics.

*Equal Function Call Contexts.* This next heuristic relaxes the uniqueness requirement of the previous heuristic and instead limits the matching only to functions with the same name to reduce the risk of false matches. More concretely, function $f_b \in \mathcal{F}_{P_b}$ and a function $f_t \in \mathcal{F}_{P_t}$ will be matched iff their $\overline{\mathcal{C}}$, $\underline{\mathcal{C}}$ and names are equal.

*Similar Function Call Contexts.* This last heuristic matches functions $f_b \in \mathcal{F}_{P_b}$ and $f_t \in \mathcal{F}_{P_t}$ with the same name and similar call contexts. We define context similarity in terms of subsets: contexts $\overline{\underline{\mathcal{C}}}_{f_b}$ and $\overline{\underline{\mathcal{C}}}_{f_t}$ are similar iff $\overline{\mathcal{C}}_{f_b} \subseteq \overline{\mathcal{C}}_{f_t} \vee \overline{\mathcal{C}}_{f_t} \subseteq \overline{\mathcal{C}}_{f_b}$ and $\underline{\mathcal{C}}_{f_b} \subseteq \underline{\mathcal{C}}_{f_t} \vee \underline{\mathcal{C}}_{f_t} \subseteq \underline{\mathcal{C}}_{f_b}$. The intuition behind this heuristic is that function call contexts may often change between versions and may contain new, or lack some previously present, callee or caller functions. To reduce the number of potential false matches, only functions with the same name are eligible for matching.

## 4.3 Heuristics Application Order

In general, we designed these heuristics to be applied in a sequence such that each subsequent heuristic tries to match the functions that still remain unmatched. Although these heuristics could be applied in an arbitrary order, we propose a specific application order: from the most strict to the most lenient ones. This way we avoid potential mismatches that would happen if the lenient, low-confidence heuristics were applied first.

The proposed ordering is as follows: 1. *Extended Function Summary Exact Matches*, 2. *Function Summary Unique Renames*, 3. *Unique Function Call Contexts*, 4. *Function Summary Exclusive Renames*, 5. *Equal Function Call Contexts*, and 6. *Similar Function Call Contexts*. Note that the heuristics are not applied in exactly the same order as they were defined.

The intuition behind this ordering is that the first three heuristics attempt to find the "low-hanging fruit" in terms of function similarity by checking for complete summary or context equality. *Function Summary Exclusive Renames* is used next as it relies on the already known renames from the second and third heuristics, and no subsequent heuristic is designed to identify renames. Also, while the *Unique Function Call Contexts* heuristic could potentially find more matches after the *Function Summary Exclusive Renames* is executed, the subsequent *Equal Function Call Contexts* heuristic will be able to identify those as well, hence no precision is lost by this specific ordering.

The last two heuristics work the best when the known renames mapping is as precise as possible, hence why they are applied only after all the heuristics that can extend the known renames mapping. Naturally, the *Similar Function Call Contexts* heuristic is potentially the least precise one, and as such is left to be applied the last. Any remaining functions not matched by any of the heuristics are either considered a low confidence name-only match if they appear in both baseline and target, or new or deleted if their name counterpart is not found in the other version.

## 4.4 Control Flow Graph Difference Analysis

Once the function bijection $m$ is constructed (or when candidate matches are being evaluated, e.g., in the *Function Summary Exclusive Renames* heuristic), the matched functions may be compared. Our FCFG diff analysis algorithm works by traversing the $FCFG_{f_b}$ and $FCFG_{f_t}$ in lockstep in a deterministic manner and comparing every pair of visited edges and basic blocks according to the equality criterion. Given our use-case for FCFG diff analysis in Perun, the algorithm simply returns whether two functions are equal or not. Moreover, we can implement this check efficiently: upon encountering the first difference we can terminate the analysis early.

The traversal starts at the entry basic block and continues to the subsequent basic block through the next yet unvisited outgoing edge. We chose the next unvisited outgoing edge according to its type: the `fallthrough` edge has the highest priority as there may be at most one such edge; while the remaining `jump` edges are ordered according to the destination address to ensure deterministic traversal of the edges. Whenever the algorithm reaches a basic block that has no remaining unvisited outgoing edge, or the basic block is already present in the current CFG traversal path[9], the traversal backtracks to the previous basic block. We keep the current CFG path in a stack structure.

We compare individual basic blocks on a per-instruction basis: the instructions are iterated from the first to last and for each instruction, its *mnemonic* and operands are compared. However, comparing the individual basic blocks for an absolute equality is generally too strict in practice, as address or register operands might change in subsequent compilations due to a number of reasons not associated with code changes in the analysed function. Hence, we compare individual basic blocks in a generic manner – the algorithm defines several comparison criteria listed from the most to least strict: (1) absolute instructions and operands equality, (2) instructions equality with register bijection (3) instructions equality with register bijection and ignoring addresses or offsets, (4) instructions equality while ignoring the operands, and (5) the number of instructions. Note that when we compare the fully unfolded destinations of call instructions, i.e., we take into account any previously detected renaming.

## 4.5 Reusing Dynamic Models

We will close our approach with a discussion on how to efficiently reuse the dynamic models. We emphasize that obtaining precise dynamic control flow models is often quite expensive as it typically entails, e.g., running the entire test suite with some form of profiling or tracing enabled. As such, when a control flow model is being constructed

[9] By CFG traversal path, we mean the sequence of basic blocks $BB_0, BB_1, \dots, BB_i$ where each pair of adjacent basic blocks $(BB_n, BB_{n+1}) \in \mathcal{C}_P$ for $0 \leq n < n+1 \leq i$.

for a new project version, we reuse parts of the dynamic models observed in previous versions of the software. This way, we get to reduce the number of dynamic runs that need to be executed in order to improve the accuracy of the static models (and, in extreme cases, possibly even skip the runs altogether if the entire dynamic model can be reused). However, we want the reuse to be conservative to not accidentally include dynamic call relations that may no longer exist in the new versions of the project.

The algorithm for identifying reusable parts of the dynamic models iterates over all matched baseline and target function pairs $(f_b, f_t) \in m$ and evaluates whether the functions satisfy at least one of these two conditions: (1) the difference analysis determined that $f_t$ has not changed compared to $f_b$, or (2) the callee contexts restricted to the static control flow models only, $\underline{\mathcal{C}}^s_{f_b}$ and $\underline{\mathcal{C}}^s_{f_t}$, are equal. If at least one condition is satisfied, the entire dynamic callee context $\underline{\mathcal{C}}^d_{f_b}$ is reused in $f_t$ as its dynamic callee context $\underline{\mathcal{C}}^d_{f_t}$. The intuition is simple: if the FCFGs or callee contexts obtained by the same static analysis tools are identical, then the dynamic call relations are highly likely[10] to be present in the new project version as well – provided the same dynamic runs are still executable in the new project version.

As the algorithm depends on the function mapping and difference analysis results, it is currently scheduled only after those analyses have finished. However, we plan to explore the possibility of running the algorithm simultaneously with the function matching and diff analysis to improve both their recall and precision.

## 5 CPython Use-case Demonstration

To demonstrate the capabilities and efficiency of the proposed heuristics and algorithms, we applied them on the CPython project; more precisely on its versions 3.10.4 and 3.11.0a7. We have selected two use-cases that demonstrate our approach.

- **UC1:** Constructing static control flow models of both CPython versions, and observing the efficiency and accuracy of both the function matching heuristics and diff analysis criteria.
- **UC2:** Comparing our matching heuristics with our implementation[11] of the matching algorithm described in [11].

We measured the results for both use-cases on a Linux Fedora 40 machine with 8 cores and 16 threads, AMD Ryzen 7 PRO 5850U 1.9 GHz CPU and 16 GB of RAM. All measurements were repeated 7 times with the first two runs being discarded as warm-up runs. Our implementation is written in Python 3.10 and we used the *Angr* [15] library for the static CFM construction. The total number of functions discovered in the CPython binary by Angr in 3.10.4 and 3.11.0a7 versions is 5262 and 5486, respectively.

[10] Nonetheless, this algorithm is generally not sound nor complete. More advanced interprocedural analyses would likely be needed to enhance the precision of this algorithm.

[11] As far as we know, the authors did not publish a software artifact with their implementation.

**Table 1.** An overview of the number of matches and renames identified by each individual heuristic. The heuristics were applied exactly in the order defined in Section 4.3.

| **Function Matching Heuristic** | $\|match\|$ | $\|rename\|$ | $\|remain_{3.10}\|$ | $\|remain_{3.11}\|$ |
|---|---|---|---|---|
| *Ext. Function Summary Exact Matches* | **3945** | 0 | 1317 | 1541 |
| *Function Summary Unique Renames* | 0 | 19 | 1298 | 1522 |
| *Unique Function Call Contexts* | 147 | 0 | 1151 | 1375 |
| *Function Summary Exclusive Renames* | 0 | 3 | 1148 | 1372 |
| *Equal Function Call Contexts* | 103 | 0 | 1045 | 1269 |
| *Similar Function Call Contexts* | **826** | 0 | 219 | 443 |
| **Total** | **5021 (95%)** | **22** | **219** | **443** |
| **Avg. runtime [s]** | | 0.1569 | | |

**Table 2.** An overview of the number of functions identified as changed, resp. unchanged, using the various diff analysis equality criteria defined in Section 4.4.

| **FCFG Diff Analysis Equality Criterion** | $\|changed\|$ | $\|same\|$ | Avg. runtime [s] |
|---|---|---|---|
| *Exact instructions and operands equality* | 4002 | 1019 | 0.3100 |
| *Exact instructions equality, register and op. bijection* | 4001 | 1020 | 0.6173 |
| *Exact instructions equality and register bijection* | **1370** | **3651** | **2.3170** |
| *Exact instructions equality* | 1363 | 3657 | 0.9184 |
| *Instruction count* | 1285 | 3736 | 0.8184 |

## 5.1 Use-case 1: Function Matching and Diff Analysis

Table 1 shows the results obtained by running the heuristics in the proposed order on CFMs constructed for CPython 3.10.4 and 3.11.0a7. We list for each heuristic the number of matched functions as $|match|$, the number of identified renames as $|rename|$ and the number of remaining functions to match in each version as $|remain_{3.10}|$ and $|remain_{3.11}|$. Overall, our heuristics successfully matched 95 % of the functions in both CPython versions and identified 22 renames out of 150 missing ($\mathcal{F}_{missing}$) and 374 new ($\mathcal{F}_{new}$) names. Functions identified as renames were manually inspected and we confirmed that they indeed represent a simple rename without a significant, that is, control flow model altering change. The remaining 219 *baseline* and 443 *target* unmatched functions represent either actually new and deleted functions, or functions that have structurally changed so much that it is unclear whether they are corresponding to their baseline or target counterpart in any other way than just by their name.

Table 2 shows the results of our FCFG diff analysis algorithm executed after the function matching step. The Table lists the different equality criteria described in Section 4.4 and the number of changed ($|changed|$), resp. unchanged ($|same|$), functions each criterion identified. We plan to build on the proposed difference analysis with these equality criteria, and possibly some more advanced tools such as DiffKemp [12], in our future work on optimised profiling. The optimised profiling relies, among other, on the ability to control the strictness of the function equality comparison within the employed function difference analysis, which is supported by our analysis approach.

**Table 3.** A comparison of identified matches, renames, false positives and runtime overhead of the proposed heuristics (divided into the summary-based and call context-based categories) with the two-step matching algorithm in [11].

| **Algorithm** | $\lvert match \rvert$ | $\lvert rename \rvert$ | $\lvert false\ matches \rvert$ | Avg. runtime [s] |
|---|---|---|---|---|
| *Proposed function summaries* | 3945 | 22 | 0 | 0.0817 |
| *Call contexts matching* | 1076 | 0 | 0 | 0.0752 |
| **Total** | 5021 | 22 | 0 | 0.1569 |
| *Function summaries in [11]* | 1703 | 9 | 20 | 1.3564 |
| *N-gram similarity matching in [11]* | 3475 | 80 | 1278 | 448.0557 |
| **Total** | 5178 | 89 | **1298** | **449.4121** |

### 5.2 Use-case 2: Comparison

In the second use-case, we compared our three summaries-based and three call context matching heuristics with the existing two-step matching algorithm from [11]. We summarise the obtained results in Table 3; although the algorithm proposed by [11] managed to overall match approximately 3 % more functions and identify almost four times more renames, it suffers from 25 % false match rate and several orders of magnitude worse run time.

The vast difference in performance and false match rate can be explained by the assumption about the connectivity of the CG made by the authors of [11]. The originally proposed algorithm consists of two steps. First, a summary-based iterative matching is used; in its initial step, it finds exact and unique summary matches, and subsequently traverses and matches the rest of the CGs based on the existing call relations $\mathcal{C}_P$. However, for partially disconnected CGs with a lot of call context changes – e.g., CPython CGs of different major versions constructed by Angr – this approach may fail and lead to many unmatched functions. Second, the computationally expensive[12] and less precise N-gram similarity matching is used to match all of the remaining functions, resulting in poor performance and high false match rate as no cut-off value for the similarity measure is defined.

## 6 Conclusion

In this work we propose (1) an efficient layered control flow models representation that conservatively reuses parts of dynamic models, (2) a hierarchical versioning system for the storage of said models, (3) six function matching heuristics tailored specifically for evolving software, and (4) a fast control flow difference analysis algorithm.

We have demonstrated our function matching heuristics and diff analysis in an use-case with CPython project that resulted in a 95 % functions successfully matched with no detected false positives tied to the function renames discovery. Although our heuristics match about 3 % less functions compared to the existing approach in [11], they are faster by several orders of magnitude and work even on partially disconnected CGs.

[12] The algorithm needs to compute similarity for every pair of unmatched baseline and target functions. Each such function comparison entails the computation and comparison of N-gram sets for every pair of baseline and target basic blocks within those functions.

**Acknowledgments.** The work was supported by the project GA23-06506S of the Czech Science Foundation, FIT-S-23-8151 of the BUT FIT, Brno Ph.D. Talent Programme, Red Hat Inc. and Horizon Europe CHESS 101087529.

**Disclosure of Interests.** Jiří Pavela and Tomáš Fiedor are, or have been in the past, funded by Red Hat, Inc. research scholarships.

## References

1. Infer: Differential Workflow, https://fbinfer.com/docs/steps-for-ci/
2. Jenkins CI performance plugin, https://jenkinsci.github.io/performance-plugin/RunTests.html
3. Chen, J., Yu, D., Hu, H., Li, Z., Hu, H.: Analyzing performance-aware code changes in software development process. In: Proc. of ICPC'19. pp. 300–310
4. Chen, J., Shang, W.: An exploratory study of performance regression introducing code changes. In: Proc. of ICSME'17. pp. 341–352
5. Chen, T.H., Nagappan, M., Shihab, E., Hassan, A.E.: An empirical study of dormant bugs. In: Proc. of MSR'14. p. 82–91
6. Fiedor, T., Pavela, J., Rogalewicz, A., Vojnar, T.: Perun: Performance version system. In: Proc. of ICSME'22. pp. 499–503
7. Fukami, C., Mccubbrey, D.: Colorado benefits management system (c): Seven years of failure. Communications of the Association for Information Systems **29**, 97–102 (2011)
8. Hilton, M., Tunnell, T., Huang, K., Marinov, D., Dig, D.: Usage, costs, and benefits of continuous integration in open-source projects. In: Proc. of ASE'16. p. 426–437
9. Humble, J.: Continuous Delivery: Evidence and case studies, https://continuousdelivery.com/evidence-case-studies/
10. Javed, O., Dawes, J.H., Han, M., Franzoni, G., Pfeiffer, A., Reger, G., Binder, W.: Perfci: A toolchain for automated performance testing during continuous integration of python projects. In: Proc. of ASE'20. p. 1344–1348
11. Lee, Y.R., Kang, B., Im, E.G.: Function matching-based binary-level software similarity calculation. In: Proc. of RACS'13. p. 322–327
12. Malík, V., Vojnar, T.: Automatically checking semantic equivalence between versions of large-scale c projects. In: Proc. of ICST'21. pp. 329–339
13. Nagarajan, V., Gupta, R., Zhang, X., Madou, M., de Sutter, B., de Bosschere, K.: Matching control flow of program versions. In: Proc. of ICSM'07. pp. 84–93
14. Rimsa, A., Nelson Amaral, J., Pereira, F.M.Q.: Practical dynamic reconstruction of control flow graphs. Software: Practice and Experience **51**(2), 353–384 (2021)
15. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: Proc. of SP'16
16. Zaman, S., Adams, B., Hassan, A.E.: Security versus performance bugs: A case study on firefox. In: Proc. of MSR'11. p. 93–102