

Patch Verification via Multiversion Interprocedural Control Flow Graphs

Wei Le, Shannon D. Pattison
Rochester Institute of Technology
One Lomb Memorial Drive, Rochester, NY, USA
{wei.le, sdp1929}@rit.edu

ABSTRACT

Software development is inherently incremental; however, it is challenging to correctly introduce changes on top of existing code. Recent studies show that 15%-24% of the bug fixes are incorrect, and the most important yet hard-to-acquire information for programming changes is whether this change breaks any code elsewhere. This paper presents a framework, called *Hydrogen*, for patch verification. Hydrogen aims to automatically determine whether a patch correctly fixes a bug, a new bug is introduced in the change, a bug can impact multiple software releases, and the patch is applicable for all the impacted releases. Hydrogen consists of a novel program representation, namely *multiversion interprocedural control flow graph (MVICFG)*, that integrates and compares control flow of multiple versions of programs, and a demand-driven, path-sensitive symbolic analysis that traverses the MVICFG for detecting bugs related to software changes and versions. In this paper, we present the definition, construction and applications of MVICFGs. Our experimental results show that Hydrogen correctly builds desired MVICFGs and is scalable to real-life programs such as *libpng*, *tightvnc* and *putty*. We experimentally demonstrate that MVICFGs can enable efficient patch verification. Using the results generated by Hydrogen, we have found a few documentation errors related to patches for a set of open-source programs.

Categories and Subject Descriptors

D.2.4 [Software/Program Verification]: Reliability

General Terms

Algorithms, Experimentation, Reliability, Verification

Keywords

Multiversion, Software Changes, Patch Verification

1. INTRODUCTION

As software becomes an essential part of our daily life, it is very important to be able to deliver new features, critical patches, refactorings or performance optimizations in a trustworthy way and in

a timely fashion. Nowadays, many software companies adopt an agile process and deliver incremental changes via short release cycles, e.g., Google Chrome and Firefox release new versions every 6 weeks [28]. Fast releases increase the communications between software companies and users but not the users' tolerance of bugs. A recent study shows that only 16% of smartphone users are willing to try a failing app more than twice [9]. System administrators are typically very careful about updating software because unstable new releases can lead to unrecoverable consequences. Importantly, if we do not assure release quality, an overwhelming number of failures can be returned after software deployment [36], diagnosing which can stall new releases [25].

Although important, it is challenging to ensure the correctness of software changes especially in a fast release setting. To introduce a change, developers need to understand existing code which may be written by other people, and the documentation can be missing or out of date. A recent study shows that the most important yet hard-to-acquire information for software changes is *whether this change breaks any code elsewhere* [51]. In fact, a 2011 Gmail bug that deleted emails for millions of users was caused by an incorrect code refactoring [1]. Studies on important open-source software found that 14.8%–24.4% of bug-fixes are erroneous [58]. Frequent releases typically imply multiple versions exist in the field, as users may have different habits to update software, or the old versions need to exist to be compatible with system dependencies; thus, we need to ensure that common changes, such as bug fixes, are not only effective for a program but also for all the versions maintained.

Traditional software assurance tools targeting single versions of programs are not scalable and flexible for verifying changes, as such analysis can take days to terminate for large software [11, 15]. Even worse, many warnings can be generated for the new version, but it is hard to determine which warnings are relevant to the changes. Program analysis targeting software changes include *impact analysis*, and its goal is to determine which statements in a program can be affected by a change. Yang et al. used impact analysis to isolate the code potentially affected by the changes and performed model checking only on the impacted code [55]. Although targeting changes, this approach is still exhaustive in that it explores all the paths impacted by the change for verification. Sometimes, the impact of a change can be large, leading to state explosion problems [3]. Person et al. generate and compare symbolic signatures from function calls to determine whether the semantics of programs have been changed between versions [40]. The comparison is done *offline* in that it first analyzes each program version respectively and then compares their analysis results. The problems of such an offline comparison are twofold. First, it redundantly detects information as the two versions share the majority of code. Second, the information used to compare program properties, such

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05...\$15.00
<http://dx.doi.org/10.1145/2568225.2568304>

as line numbers and variable names, can be changed between versions and hard to match.

The goal of our work is to design program analyses targeting program changes as well as multiple versions for patch verification (in this paper, we use the terms *patch* and *software changes* interchangeably). Specifically, we aim to automatically determine whether a patch fixes a bug and whether a software change breaks existing code and introduces new bugs. Importantly, we not only verify the bug fix for a program but also determine whether the fix is applicable for all the buggy releases. We design a framework, called *Hydrogen*, consisting of a program representation, namely *multiversion interprocedural control flow graph (MVICFG)*, that specifies the commonalities and differences of control flow for multiple program versions, and a demand-driven, path-sensitive symbolic analysis on the MVICFG for detecting bugs in program changes and multiple versions.

Intuitively, an MVICFG is a union [27] of a set of *Interprocedural Control Flow Graphs (ICFGs)* for program versions. Depending on applications, the program versions in an MVICFG can be revisions from code repositories or software releases. In an MVICFG, a node is a program statement, and we specify one node for multiple versions if the statement is not changed across the versions. An edge specifies the control flow between the statements. Both the nodes and edges are annotated with which versions they belong to.

To build an MVICFG, we first construct an ICFG for a program version and then incrementally integrate control flow changes for successive versions. Using an MVICFG to verify a patch, we apply an interprocedural, demand-driven, path-sensitive, symbolic analysis on the MVICFG. The analysis takes versions marked on the edges into consideration and performs either incrementally on the program changes for detecting bugs in changes, or simultaneously on multiple program versions for determining bug impact and verifying patches for multiple software releases.

The novelty and importance of an MVICFG are as follows.

1. **Representing program semantic differences:** MVICFGs are control flow based representations, and we can easily obtain the changed program paths, and thus program behaviors, from the graphs for visualization and analysis.
2. **Enabling efficient, precise program verification:** Precisely identifying bugs requires a prediction of program runtime behaviors. We apply an interprocedural, path-sensitive, symbolic analysis on the MVICFG for precision. We achieve efficiency by 1) only directing analyses along the changed program paths, 2) caching and reusing intermediate results from analyzing older versions for a new version, and 3) applying a demand-driven algorithm to traverse the changed paths only relevant to the bugs. The three approaches improve the scalability of analyses without compromising the precision, and have a great potential to be useful in practice.
3. **Correlating multiple program versions:** Using MVICFGs, the analysis not only can traverse along program execution paths but also longitudinally across program versions for comparing and sharing analysis results. Therefore, we not only can compare programs at code level but also can determine the commonalities, differences or changes of program properties (e.g., bugs or invariants) across program versions.
4. **Facilitating online comparisons:** In an MVICFG, program versions are matched based on their statements. We thus can determine the commonalities of program properties by analyzing the shared code. Meanwhile, using the matched state-

ments, we can easily report the differences between program versions. As we mentioned before, in an offline comparison, we may repeatedly analyze the same code existing in many versions and have difficulties to match the generated results from different versions.

We implemented Hydrogen using the Microsoft Phoenix infrastructure [41]. We experimentally demonstrate that our algorithm is scalable to build MVICFGs for real-life programs such as *libpng*, *tightvnc* and *putty*. We randomly selected functions from the benchmark programs and manually validated the correctness of the MVICFG. Our experiments show that the integration of demand-driven, path-sensitive, symbolic analysis and the MVICFG is feasible and efficient for detecting bugs in changes; and we are able to perform patch verification for multiple versions of software releases. Before, such information has to be identified manually. In fact, our experimental results show that such documentation can be buggy or incomplete. Note that in this paper, we mainly focus on applying static analyses on MVICFGs for bugs such as integer overflows, buffer overflows and null-pointer dereferences; however, the MVICFG is a representation that may be generally applicable for a variety of program analyses, e.g., concolic testing, for other types of bugs.

In summary, the contributions of the paper include:

- Definition of an MVICFG,
- The algorithm for constructing an MVICFG,
- Applications of an MVICFG for patch verification, and
- Implementation and experimental results to demonstrate the scalability and correctness of building MVICFGs and the effectiveness of applying MVICFGs to solve a set of important problems in patch verification.

The rest of the paper is organized as follows. In Section 2, we provide an overview of the MVICFG using an example. In Sections 3 and 4, we present the definition and the construction of an MVICFG respectively. In Section 5, we present the integration of a demand-driven, path-sensitive symbolic analysis on the MVICFG for a set of important applications. In Section 6, we describe our implementation and experimental results, followed by the related work in Section 7 and conclusions in Section 8.

2. AN OVERVIEW

In Figure 1, we use a simple example to intuitively explain the MVICFG and how to construct and use it for patch verification. As shown in Figure 1(a), versions 1–4 describe a bug fix scenario in the *FreeBSD* code repository [58]. In version 1, a buffer overflow exists at line 3. Version 2 introduces a fix by replacing the stack buffer with dynamically allocated heap memory; however, this patch does not correctly fix the bug. Version 3 enhances the code by checking an exceptional condition, but the code fails to drop the privilege along the exceptional path, leading to a privilege elevation vulnerability [13]. This bug is fixed in version 4 by adding a call *drop_privilege* at line 7. Version 4 also finally fixes the buffer overflow originated from version 1.

In Figure 1(b), we show the MVICFG constructed for the 4 versions of programs. In the graph, the statements common across versions, e.g., nodes 1, 3 and 4, are only specified once. Nodes 1, 5, 7 and 9 in solid indicate the beginning of the differences. As an example, node 1 leads the differences between version 1 and versions 2–4. Edges are labeled with versions (in Figure 1(b), we

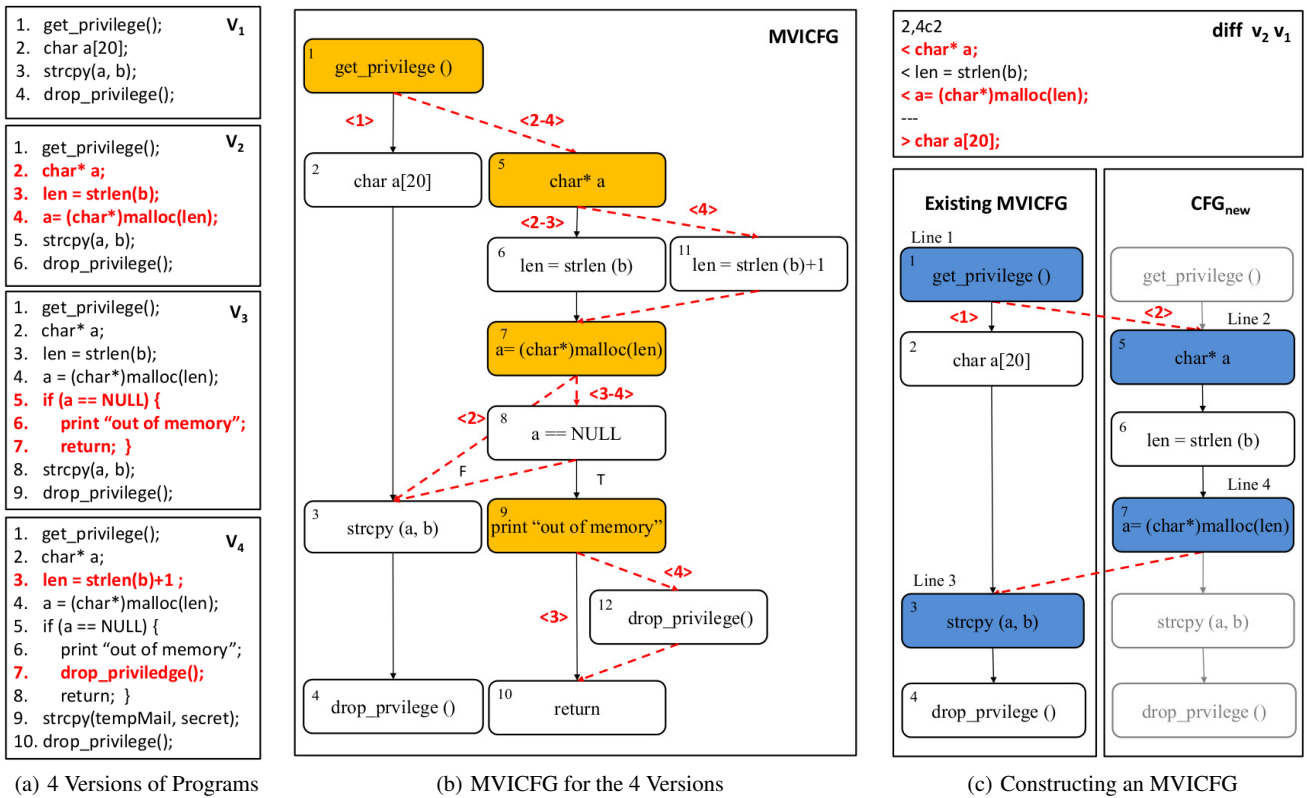


Figure 1: Four Versions of Programs and their MVICFG

only mark the edges connected to the beginning of the differences for clarity). The edges, such as $\langle 1, 5 \rangle$, that connect changes to the new versions are specified in the dotted line in the graph. From the graph, we can obtain the changed program paths for a new version. For example, in Figure 1(b), starting at node 1 and following the versions marked on the edges, we derive that path $\langle 1, 5 - 7, 3, 4 \rangle$ is newly added in version 2.

To build an MVICFG shown in Figure 1(b), we first construct an ICFG for version 1 and then incrementally add the control flow differences for the next versions. Figure 1(c) displays the process to construct an MVICFG consisting of versions 1 and 2. In the first step, we identify which statements are different between the two versions, shown on the top. Based on the statement differences, we find the corresponding nodes 1, 3, 5 and 7 on the graphs representing the entries and exits of the differences. In the next step, we connect node 1 in version 1 to node 5 in version 2 and node 3 in version 1 to node 7 in version 2 and update the versions on the edges, shown at the bottom in Figure 1(b).

We develop a demand-driven, path-sensitive symbolic analysis on the MVICFG to verify bug fixes and detect bugs introduced by a change. In this example, to determine if the patch in version 2 correctly fixes the buffer overflow, we raise query $[\text{size}(a) > \text{len}(b)]$ at node 3 in Figure 1(b). The query is propagated backwards along path $\langle 3, 7, 6 \rangle$ in version 2 and resolved as $[\text{len}(b) > \text{len}(b)]$ at node 6, indicating an off-by-one buffer overflow exists along path $\langle 1, 5 - 7, 3 \rangle$. Thus, the bug is not fixed correctly. To detect bugs in the change added in version 3, we first apply a reachability analysis from nodes 8–10 introduced in version 3 and determine that path $\langle 1, 5 - 10 \rangle$ is new to this version. The demand-driven analysis starts at *get_privilege* at node 1, inquiring a liveness property regarding whether a *drop_privilege* will be called after *get_privilege*.

At node 10, we discover *drop_privilege* is never called along the new path $\langle 1, 5 - 10 \rangle$, leading to a privilege elevation.

The MVICFG also can be used to determine whether a bug can impact multiple software releases and whether a patch developed based on a version can fix the bug for all the impacted releases. To explain the use of MVICFGs in this scenario, we assume versions 1–3 in Figure 1(a) are the three deployed software releases, and the buffer overflow in version 3 is reported by the user. To determine which other versions the bug may impact, our analysis raises query $[\text{size}(a) > \text{len}(b)]$ at node 3 in Figure 1(b), aiming to check, for all the paths of versions 1–3 reachable from node 3, whether the buffer overflow exists. The query is resolved at node 2 along path $\langle 3 - 1 \rangle$ in version 1 and also at node 6 along paths $\langle 3, 7, 6 \rangle$ in version 2 and $\langle 3, 8 - 6 \rangle$ in version 3, indicating the bug can impact versions 1–3. To patch the bug, a common practice is to diagnose and introduce the fix based on one impacted version. Suppose, in this case, we develop a patch based on version 3 where the bug is firstly discovered, and the patch removes node 6 and adds node 11, shown in Figure 1(b). Our analysis aims to determine whether the patch developed for version 3 can also fix the buffer overflow in versions 1–2. From the graph, we see that version 1 does not contain node 6, and thus the patch cannot be directly applied. Similarly, we found that the patch can be applied to version 2 without leading to a syntax error; however, further semantic analysis needs to be applied to confirm whether the buffer overflow in version 2 is removed with this patch. To do so, we propagate query $[\text{size}(a) > \text{len}(b)]$ at node 3 along all the paths of versions 2–3. At node 7, we arrive at the patch location, we advance the query to node 11 instead of node 6 to integrate the patch. The query is resolved at node 11 as $[\text{len}(b)+1 > \text{len}(b)]$, indicating the buffer access is safe; that is, the patch correctly fixed versions 2 and 3.

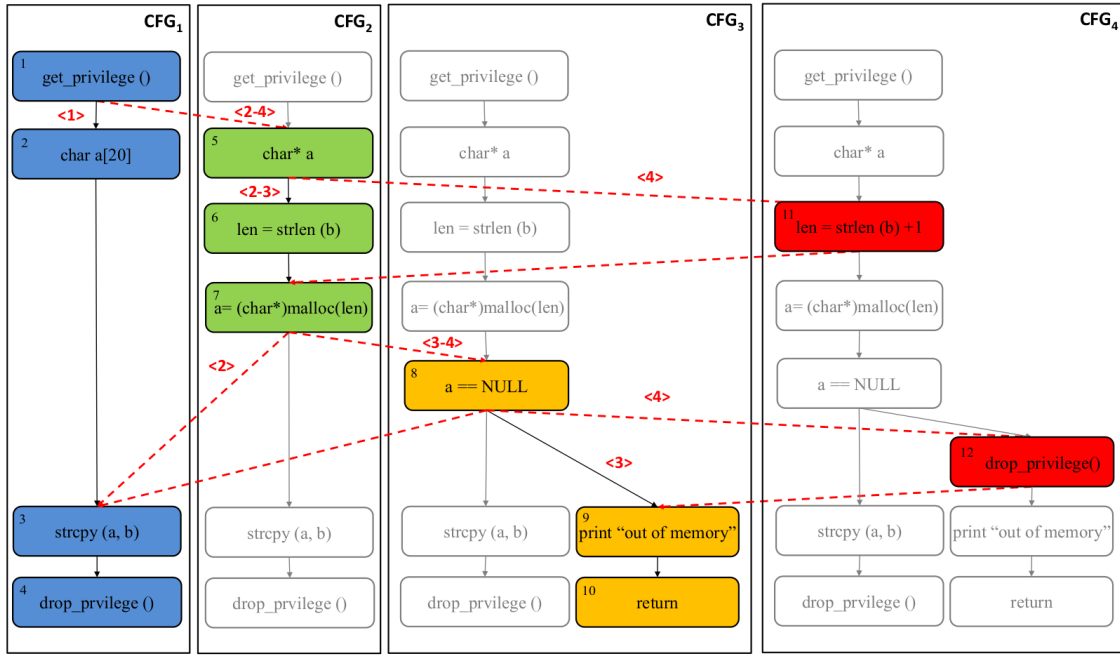


Figure 2: MVICFG: the Union of ICFGs

3. DEFINING AN MVICFG

We aim to design the MVICFG that can satisfy the following requirements: 1) it integrates control flows of n versions of programs of interest, and the differences and commonalities of the control flow between any versions are available on the graph; 2) from the MVICFG, we are able to get a subgraph of the MVICFG that represents any desired m ($1 \leq m \leq n$) versions of programs; and 3) from the MVICFG, we are still able to obtain the control, data and value flow as well as variable dependencies that originally belong to each individual version of a program. With the above goals, we present our definition for MVICFGs.

Definition: An Multiversion Interprocedural Control Flow Graph (MVICFG) $G = \langle N, E \rangle$ is a union of G_1, \dots, G_k , such that $G_i = \langle N_i, E_i \rangle$ is an ICFG representing the i^{th} version of program P_i . $n \in N_i$ is a statement in P_i . $\langle n, m \rangle \in E_i$ is an edge in P_i . $\forall n \in N_i, n \in N$. $\forall \langle n, m \rangle \in E_i, \langle n, m \rangle \in E$. For n matched across versions, we label $V_N(n)$ to denote the set of program versions to which n belongs. Similarly, we label edge $\langle n, m \rangle$ with $V_E(\langle n, m \rangle)$ to denote the set of program versions to which the edge belongs.

Example: In Figure 2, we show that the MVICFG in Figure 1(b) is a union of the ICFGs of the 4 program versions given in Figure 1(a). In each ICFG, the nodes newly introduced in the current version are specified in solid, and the nodes matched to previous versions are shown in Grey. The *match* is determined by the programmers' beliefs on whether the statement is changed from the previous version to the current version. For example, in Figure 2, node 1 is matched across 4 versions, as from the scenario in Figure 1(a), we believe that the statement at line 1 has not been changed for the 4 versions. Using this way, we only specify once for the nodes and edges commonly shared across versions. Since we annotate nodes and edges with the version numbers, the control flow information is not lost in the MVICFG when we perform a union for a set of ICFGs [27].

4. CONSTRUCTING AN MVICFG

Here, we present our approach for constructing an MVICFG.

4.1 An Overall Approach

To construct an MVICFG, we identify what are the common nodes between versions, and we then incrementally add the control flow changes from new versions on top of the in-progress MVICFG. Shown in Figure 3, we take 5 steps to integrate a new version to an MVICFG. In the first step, we identify the differences between the n^{th} version, the version we aim to integrate, and the $n - 1^{\text{th}}$ version, the last version integrated on the in-progress MVICFG. To do so, we first determine whether in a new version, a function is added, removed or updated. For an updated function, we report which statements are added and/or removed. We choose to compare the functions of the two versions statement by statement rather than line by line directly from the source code, because in a CFG, each node is a statement, and the added and/or removed statements identified from this step will be used in the next step to construct control flow that can be integrated to the MVICFG.

In the second step, we identify the change of control flow between the two versions. Our approach is to first build CFGs for the newly added code as well as the code after deletion. Next, we find where on the MVICFG, the new statements should be added, and we connect the MVICFG nodes to the corresponding entries and exits of the new code. Similarly, we identify on the MVICFG which nodes are deleted. We then compare the CFG of the new version to adjust the corresponding MVICFG edges. Our final step is to update the version information for all the edges and nodes in the updated functions.

4.2 The Algorithm

We provide further details on how to construct an MVICFG in Algorithm 1. The input of the algorithm is n versions of source code, and the output is an MVICFG. At line 1, we first build an ICFG for the first version. Lines 2–11 handle a version a time and

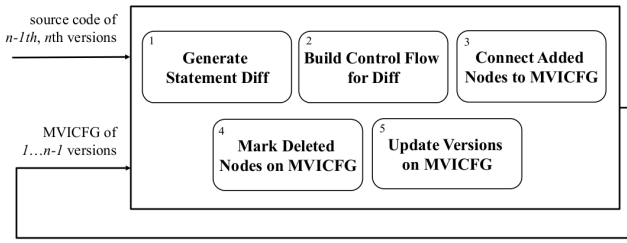


Figure 3: Building MVICFGs

ALGORITHM 1: Construct an MVICFG for n Program Versions

Input : n versions of source code v_1, v_2, \dots, v_n
Output: the mvicfg that integrates v_1, v_2, \dots, v_n

```

1 mvicfg = BuildICFG( $v_1$ )
2 foreach new version  $v_i$  do
3   D = GenerateDiffs( $v_i, v_{i-1}$ )
4   foreach diff  $\in D$  do
5     if add or update a function then  $cfg = BuildCFG$ (diff.func);
6     if add statements then
7       AddToMVICFG( $cfg, mvicfg, diff$ );
8     if delete statements then
9       DeleteFromMVICFG( $cfg, mvicfg, diff$ );
10    end
11  end
12 Procedure AddToMVICFG( $CFG\ cfg, CFG\ mvicfg, Diff\ d$ )
13  N = IdentifyAddedNodes( $cfg, d$ )
14  foreach  $n \in N$  do
15    T = Pred( $n, cfg$ )  $\cup$  Succ( $n, cfg$ );
16    foreach  $t \in T$  and  $t \notin n$  do
17       $t' = FindMatchedNode$ ( $t, d, mvicfg$ ); AddEdge( $t', n$ );
18    end
19    UpdateVersion( $n$ );
20  end
21 Procedure DeleteFromMVICFG( $CFG\ cfg, CFG\ mvicfg, Diff\ d$ )
22  N = IdentifyDeletedNodes( $mvicfg, d$ );
23  foreach  $n \notin N$  and  $n \in mvicfg$  do
24    if  $n$  has a successor or predecessor in  $N$  then
25       $n' = FindMatchedNode$ ( $n, d, cfg$ );
26      foreach  $m' \in Pred(n', cfg) \cup Succ(n', cfg)$  do
27         $m = FindMatchedNode$ ( $m', d, mvicfg$ );
28        if no edge between  $m, n$  on  $mvicfg$  then AddEdge( $n, m$ );
29      end
30    end
31    UpdateVersion( $n$ );
32  end

```

incrementally integrate the differences from the next versions.

GenerateDiffs at line 3 accomplishes two tasks. First, it determines whether in a new version, a function is added, removed or updated by comparing function signatures from the two versions. A function is added if the function signature only appears in the new version; similarly, a function is deleted if the function signature only exists in the old version. If the function signature has not been changed but the statements in the function body are different, we consider the function is updated. For an updated function, we further identify which statements are added and/or deleted in the new version. To obtain statement differences, we use a parser to generate a text file for each function, where each line contains a statement. We then use a UNIX diff tool [26] to compare the two textual files to determine which statements are added and/or removed in the function in version n .

BuildCFG at line 5 constructs control flow graphs for the updated code in the new version. Theoretically, we just need to construct

control flow for the code involved in the change. In practice, we have not found a tool that can build a CFG for any selected section of statements. Thus, in our implementation, we build a CFG for the complete updated function and mark the added and removed nodes on the CFG.

AddToMVICFG at lines 12–20 takes cfg , the control flow graph built for the updated function, $mvicfg$, the in-progress mvicfg built for the previous versions, and d , the statement differences, including both added and removed statements in a new version of the function. The goal is to identify the control flow for newly added code from cfg based on d and append it to $mvicfg$. At line 13, we first identify on cfg the set of nodes that are newly added—these nodes should be linked in to $mvicfg$. To find their successors and predecessors on $mvicfg$, our approach is to first find their successors and predecessors on cfg (see lines 15–16). We then map these entries and exits of the differences to $mvicfg$ (see line 17) and connect them to the new statements in cfg . At line 19, we update the version information for the nodes and edges related to the change. *UpdateVersion* at line 19 records in which version a node in the MVICFG is introduced and removed; that is, which versions the nodes and edges belong to.

To handle deletion, we do not need to add or remove nodes from the in-progress MVICFG. Instead, we just need to update the edges and versions that can reflect the control flow change. Details are given in *DeleteFromMVICFG* at lines 21–32. At line 22, we first detect N , the set of deleted nodes on $mvicfg$. At lines 23–24, we find the predecessors and successors of the deleted nodes on $mvicfg$ and determine if any edge needed to be adjusted regarding these nodes for representing the new version. To do so, we map these predecessors and successors on $mvicfg$ to cfg at line 25 and find their corresponding edges on cfg , if any, at line 26. If we find such edges, at line 28, we add them on $mvicfg$. At line 31, we traverse the deleted nodes and edges to update the versions.

We believe that Algorithm 1 builds an MVICFG that can satisfy the requirements and definition shown in Section 3. First, in our approach, we classify software changes into adding, removing and updating a function, and for the updated function, we further classify whether the update is an addition and/or removal of the statements. If a function is added or removed in a new version and it is not dead code, the caller(s) of this function is surely updated by adding or removing a callsite of this function. Thus, after building CFGs for the newly added function (see line 5 Algorithm 1), the case of adding or removing a function can be reduced to how to integrate an updated function in the MVICFG. For the two types of changes, adding and removing statements, we handle addition at line 7 and deletion at line 9 in Algorithm 1.

Second, the MVICFG is defined by the nodes and edges as well as the versions marked on these nodes and edges. Although in the implementation, we built CFGs for the complete updated functions rather than only for the changed code, only the nodes and edges labeled with version information are linked into the MVICFG.

Third, we use statement differences to determine control flow differences between versions. Although statements might have an ambiguous definition and may mean differently in different languages, the key is that the statements we used to report the differences are the same as the statements constructed in our CFG nodes.

Finally, to determine the *match* between the nodes in program versions, we use a UNIX diff tool to detect the differences between statement sequences. The tool implements a longest common subsequence algorithm [26]. Our assumption is that the differences detected by such algorithm are consistent with the programmers' intention on which statement is changed.

5. APPLICATIONS OF AN MVICFG

The MVICFG is a control-flow based program representation for specifying software changes and comparing program versions. The goal is to extend program analysis applicable on ICFGs to MVICFGs and determine program properties related to program changes and versions. Here, we present an integration of a demand-driven, path-sensitive, symbolic analysis with the MVICFG for a set of important tasks in patch verification.

5.1 Demand-Driven, Path-Sensitive Analysis

Demand-driven analysis formulates a demand into queries about program facts. We thus can determine program properties by performing a traversal of a program that is only relevant to resolving the query for efficiency. Demand-driven analysis is a natural fit for the patch verification problem because not only bugs but also program changes are likely to be sparsely distributed in the code. Not every execution path in a program or every statement along the path is relevant to the bugs or changes. Therefore, a search from the program points of interest of bugs and changes in a demand-driven fashion on the parts of the paths that are relevant may greatly improve the scalability of the analysis. With the efficiency achieved, we then potentially afford more precise, expensive analysis, such as interprocedural, context-sensitive, path-sensitive, symbolic analysis, to reduce false positives and false negatives of bug detection.

Demand-driven analysis formulates bug detection to queries at *potentially faulty statements (PFS)*, where a faulty condition, e.g., a buffer overflow, can be perceived [35]. We have identified for a set of common bugs, the types of PFSs and the queries that can be raised at these statements [35]. For instance, to detect buffer overflow, we raise a query at the buffer access inquiring whether the buffer safety constraints can be violated. To detect the bug, we traverse all the paths to this buffer access to resolve the queries. Applying to detect bugs in changes, we only need to find PFSs along the changed program paths identified on the MVICFG.

Path-sensitive analysis tracks program facts along actual program paths. Path-sensitive analysis is potentially precise to predict runtime program behavior because 1) it does not merge queries from different paths, and 2) it excludes any statically detectable infeasible paths. Unlike on ICFGs where any path in the graph represents a program path, on an MVICFG, the edges along a path may belong to different versions of programs. Thus the challenge of applying path-sensitive analysis on the MVICFG is to ensure the propagation of program facts is always along the edges of the same program versions.

Demand-driven, path-sensitive, symbolic analyses have been successfully applied for detecting bugs for single versions of programs [33, 34, 35, 32]. We thus take the design decisions of handling loops, procedures and infeasible paths shown to be scalable and effective for analyzing single versions of programs to design analyses on MVICFGs. Specifically, to handle loops, we first traverse a loop once to determine if the loop has an impact on the query, and if so, what is the symbolic change for the query through one iteration. Meanwhile, we identify the symbolic loop iterations if possible. If both the loop impact and iterations are successfully identified, we then determine the symbolic update of a query in the loop. Driven by the demand, we only need to reason the loops that can have an impact on the query. For handling procedural calls, our approach is an interprocedural, context-sensitive analysis. In a backward demand-driven analysis, we propagate the query to its original caller at the entry of the procedural call. Our interprocedural analysis is also demand-driven in that we only propagate a query into a procedure from the callsite if it is determined to have an impact on the query. To reduce the false positives caused by in-

feasible paths, we first identify infeasible paths based on a branch correlation algorithm and mark them on the MVICFG [8]. We only propagate the query along feasible paths during bug detection.

5.2 Automatic Patch Verification

We apply demand-driven, path-sensitive analyses on MVICFGs for a set of patch verification tasks. First, we develop *incremental analysis* targeting differences between any specified versions for verifying bug fixes and detecting bugs in changes. Second, we develop *multiversion analysis* that can simultaneously analyze a set of program versions to determine which software releases a bug may impact and whether a bug fix introduced for a version can potentially correct other impacted versions.

5.2.1 MVICFGs for Incremental Analysis

Incremental analysis is designed for the following two tasks.

Detecting Bugs in Changes: We need to quickly verify program changes in two scenarios. In one scenario, developers finish some changes and want to know if their code introduces new bugs and breaks the existing program. In this case, the code server can maintain the most up-to-date MVICFG for previous revisions. When the code is checked in, we incrementally integrate the new code on the MVICFG and invoke the analysis to detect bugs along the changed program paths. In another case, developers are merging a set of changes to a branch or mainline in the code repository. The analysis aims to detect semantic merge conflicts that a compiler is not able to report, e.g., the merge can cause a memory leak. We will construct an MVICFG consisting of revisions before and after the changes. Developers have flexibility in choosing how much change to be verified by selecting appropriate revisions from the code repository. In this paper, we focus to design static analysis on MVICFGs for detecting bugs such as null-pointer dereferences, buffer overflows, integer overflows and memory leaks. These bugs can cause programs to crash and thus are very important to be found at the early stage of the software development.

Verifying Bug Fixes. A certain type of code changes focuses to correct a bug. Typically, it is urgent to release such bug fixes, and thus we need to quickly verify whether the patch actually fixes the bug. As opposed to detecting bugs in changes, in this case, we know where in the code the bug is located. The goal of our analysis is to show that integrating the patch, the bug no longer exists. Instead of analyzing the changed paths for bugs, we start at the PFS where the bug is given and verify if the safety constraints for the bug can be satisfied. Note that sometimes the patch ships the bug fixes with other functionality enhancement. Using this approach, we still can quickly verify if the patch correctly fixes the bug.

A key step for incremental analysis is to raise a query at a PFS and determine its resolutions along the changed paths for bugs; that is, on the MVICFG, the analysis needs to propagate a query along a particular version. In Figure 4(a), we select nodes 2, 3, 5–8 from Figure 1(b) as an example to explain how we perform incremental analysis to verify whether version 2 fixes the buffer overflow in version 1. Shown in Figure 4(a), we raise a query at node 3, and as a part of the query, we include which versions the query aims to propagate. At node 3, the query can be advanced to nodes 2, 7 and 8. To determine which edge is legitimate, we compare the versions marked on the edges $\langle 3, 2 \rangle$, $\langle 3, 7 \rangle$ and $\langle 3, 8 \rangle$ with the version(s) stored in the query. Specifically, we perform an intersection for the set of versions from the edges and the one tracked in the query; if the intersection is not an empty set, we propagate the query onto the edge. Using this approach, the path $\langle 3, 7, 6 \rangle$ is selected for verifying the bug fix for version 2.

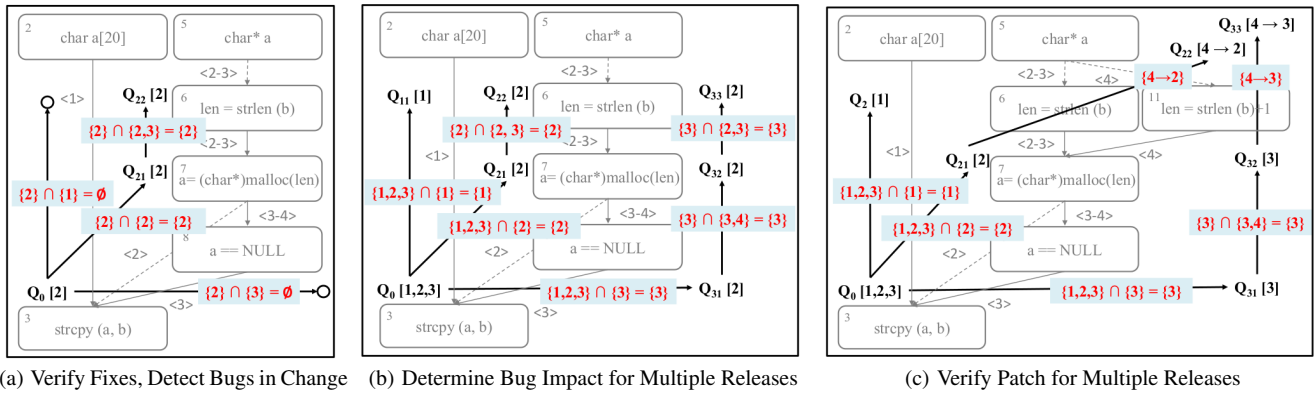


Figure 4: Select Paths on an MVICFG based on Versions

On an MVICFG, we can use three approaches to determine the bug for the change: 1) *change-based, exhaustive* analysis, similar to the one implemented in the Yang et al.’s incremental model checking [55], 2) *change-based, demand-driven* analysis, the basic demand-driven algorithm we explained above, and 3) *cache-based, demand-driven* analysis, a further optimized approach we develop for the Hydrogen framework. In Figure 5, we provide more details for the three approaches using an example in Figure 1. Suppose our goal is to verify if the patch in version 4 can correct the buffer overflow in version 3. In a change-based, exhaustive analysis shown in Figure 5(a), we would first identify paths $\langle 1, 5, 11, 7, 8, 3, 4 \rangle$ and $\langle 1, 5, 11, 7 - 9, 12, 10 \rangle$ as being impacted by the change introduced at node 11. We then start the analysis at node 1 and exhaustively collect information at each node along the two paths until node 3 is reached. In the worst case, we may have visited nodes 1, 5, 11, 7, 8, 3, 9, 12 and 10 shown in Figure 5(a), and even in the best case, we need to visit nodes 1, 5, 11, 7, 8 and 3.

The change-based, demand-driven analysis shown in Figure 5(b) would identify that path $\langle 1, 5, 11, 7, 8, 3, 4 \rangle$ is impacted by the change and contains the PFS for a potential buffer overflow. The analysis starts at node 3 and collects information in a demand-driven fashion along $\langle 3, 8, 7, 11 \rangle$. At node 11, the query resolution is determined, and thus the analysis is terminated. Note that to detect bugs in changes, we would raise queries for all the PFSs along the impacted paths. Since the query may be resolved by only propagating along the old code of the path, the change-based analysis is not most economical for only focusing on changes.

The most efficient analysis is the cache-based, demand-driven analysis shown in Figure 5(c). Here, we cache and reuse all the intermediate results obtained from analyzing previous versions on an MVICFG. When analyzing a new version, we raise queries at PFSs located in the new code. Meanwhile, we advance the cached queries previously computed at the interface of the old and new code for new resolutions. For example, in Figure 5(c), query $[\text{value}(\text{len}) > \text{len}(\text{b})]$ has been propagated to node 7 when analyzing version 3. To verify the bug fix in version 4, we can directly advance this intermediate result to node 11 to determine the correctness of the buffer access at node 3 rather than restarting the analysis from node 3. As a result, we only visit node 11 for verifying the bug fix in version 4.

5.2.2 MVICFGs for Multiversion Analysis

Here, we explain multiversion analysis for patch verification tasks related to multiple software releases.

Determining Bug Impact for Multiple Releases. We will re-

port bug impact as to which releases a bug can affect. Knowing which versions are impacted, we can determine how to develop the fixes and also notify the affected users to patch the bugs on time. Our approach is to first construct an MVICFG consisting of a set of software releases. On the MVICFG, we find the PFS where the bug is located, from which, we simultaneously analyze the paths of all versions reachable to this PFS to determine the bug impact. In Figure 4(b), suppose the bug is reported in version 3 at node 3. Our interest is to determine which other versions this bug may impact. We start the analysis at node 3 and propagate the query along paths of all versions. During the query propagation, we compare the versions stored in the query with the versions marked on the edges and only advance the query if the intersection of the two sets is not empty, shown in Figure 4(b). By doing so, we make sure the query is propagated along legitimate paths of the same versions.

Verifying Patches for Multiple Releases. Different projects may have different conventions to patch their programs dependent on their branch structures. A typical practice [2] is as follows. A reported failure is first dispatched to the owner of the code. The developer diagnoses the failure and develops the patch based on the program version where a failure occurred. The patches are then merged to the mainline or the relevant branches where a bug is affected. Determining whether a patch can fix a bug for all the affected versions is challenging because a semantic bug, such as buffer overflow, is not a local property. Even though a local function is never changed across versions and the patch is merged to all the versions without a syntax error, it does not mean the bug fix is generally applicable for all the impacted versions. We need an interprocedural, semantic analysis for further confirmation.

To verify a patch for multiple releases, we first integrate the patch to the MVICFG that consists of a set of releases. We then determine for each release whether the bug is successfully fixed after integrating the patch. In Figure 4(c), versions 1–3 contain a buffer overflow at node 3. Suppose a patch is developed based on version 3, which removes node 6 and adds node 11. First, from the graph, we find that the patch is not reachable from any paths of version 1, and thus it is not directly applicable to version 1. In the next step, we determine whether the bug in versions 2 and 3 can be fixed by this patch. We raise the query at node 3 and propagate it along paths of versions 2–3. At node 7 where the patch is encountered, we continue advancing the query to the patched code, node 11, rather than node 6, the old code that belongs to versions 2–3. Notations $\{4 \rightarrow 2\}$ and $\{4 \rightarrow 3\}$ in the figure mean at node 11 we use the patched code in version 4 to replace node 6 for determining the bug. At node 11, we resolve the query and determine the buffer overflow is fixed.

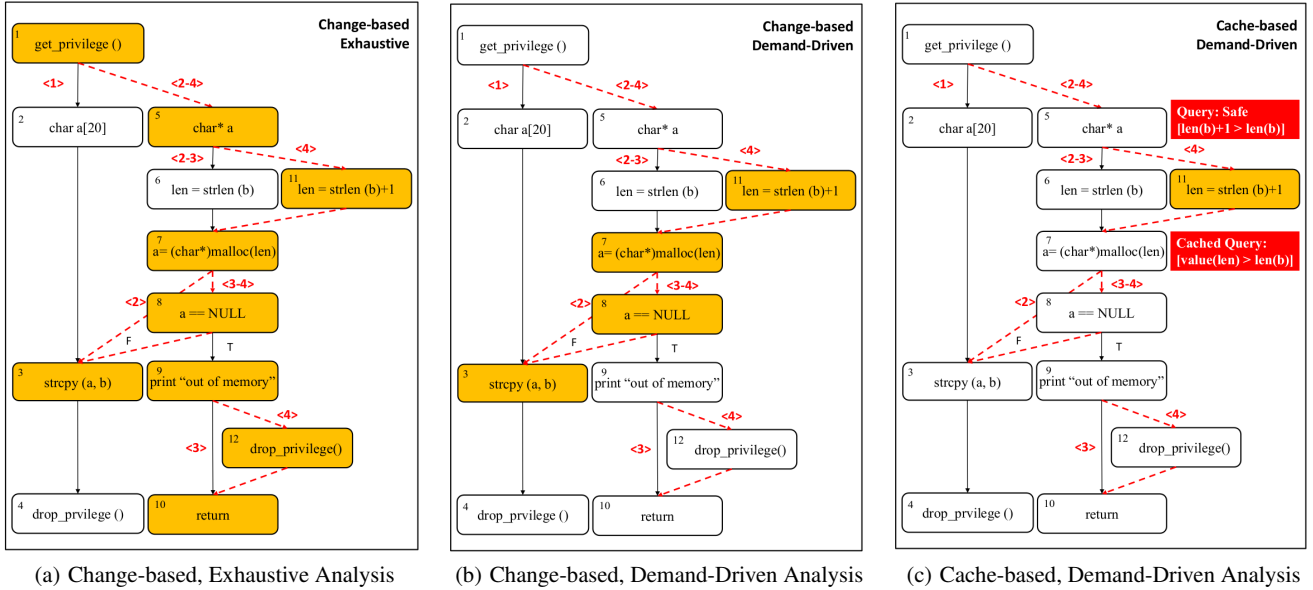


Figure 5: Efficiency via Caching Intermediate Results on the MVICFG: Comparing Nodes Visited in Three Types of Analyses

6. EXPERIMENTAL RESULTS

The general goals of our experiments are to demonstrate that an MVICFG can be correctly built and the algorithm is scalable for real-life programs and also to show the MVICFG is practically useful for a set of important tasks of patch verification.

6.1 Implementation and Experiment Setup

Hydrogen consists of two components: the construction and the analysis of an MVICFG. We implemented both of the components using the Microsoft Phoenix infrastructure [41], and we used the Microsoft Dsolver [22] as a constraint solver for bug detection. Thus, Hydrogen can handle C/C++/C# programs. To generate the differences between versions, we first preprocessed the source code. We then used srcML [50] to parse the programs for functions and applied the UNIX diff [52] on the functions to determine the added and removed statements in a new version.

We collect a set of 7 benchmarks, each of which contains multiple versions of programs. *tcas*, *schedule*, *gzip* and *printtokens* are obtained from the *sir* [49] benchmark, and programs for *libpng*, *tightvnc* and *putty* are revisions and releases selected from the real-world code repositories.

We designed three experiments. In the first experiment, we measured the time and memory used to build an MVICFG to determine the scalability of the MVICFG construction. In the second experiment, we found a set of known bugs and their patches from the Common Vulnerability Exposure (CVE) [14], Bugzilla [10] as well as the projects' revision histories. We ran Hydrogen to determine whether these patches correctly fixed the bugs. Furthermore, we randomly selected a set of releases and determine whether the bugs can impact these versions, and if so, whether the patch can also be applied to correct these impacted versions. In both of the cases, we compare our results with the documentation. In the third experiment, we demonstrate our capabilities in detecting bugs in the changes, and we compare change-based and cache-based demand-driven analyses to demonstrate the efficiency of our approach. In the following, we provide detailed results for the three experiments.

Table 1: Scalability of Building MVICFGs

Benchmark	V	LOC	Churn	ICFGs	MVICFG	T(s)	M (mb)
tcas	40	81	4.5	6.6 k	476	2.8	43.4
schedule	9	134	6	2.4 k	298	3.6	43.0
printtokens	7	198	3	2.7 k	413	0.5	43.3
gzip	5	5.0 k	242	6.8 k	2.1 k	15.1	60.9
tightvnc	5	6.3 k	457.3	10.3 k	2.4 k	30.3	106.1
libpng	9	9.2 k	1.4 k	35.3 k	8.4 k	90.1	128.0
putty	5	34.5 k	8.2 k	28.3 k	13.3 k	1844.5	310.5

6.2 Experimental Results

6.2.1 Building MVICFGs

Table 1 presents the data collected from constructing MVICFGs. Under *V* and *LOC*, we show the number of program versions used and the size of the first version in the benchmarks. Under *Churn*, we report the average number of added and removed statements between any two consecutive versions. Comparing columns *ICFGs*, the total number of nodes on the ICFGs for all the versions, and *MVICFG*, the number of nodes in the MVICFG, we show an MVICFG is able to identify the commonalities across the versions and thus largely reduce the redundancies in representing control flow for a set of programs. We report the time and memory overhead under *T* in terms of seconds and *M* in terms of megabytes. The results are collected from dual Intel Xeon E5520 CPU processors running at 2.27 GHz with 12.0 GB of RAM. We show that building MVICFGs is scalable for real-life programs. For example, it only takes 1.5 minutes to build the MVICFG for 9 versions of *libpng*. It takes about half an hour to handle *putty*. We found much time has been spent on reading pre-build CFGs of different versions from files, as Phoenix cannot build CFGs for multiple versions in a single run. To determine the correctness of the MVICFG, we have performed manual validation on the selected parts of MVICFGs.

6.2.2 Determining Bug Impact and Verifying Fixes

In Table 2, we demonstrate the usefulness of the MVICFG in determining bug impact and the correctness of the bug fixes. We

Table 2: Determining Bug Impact and Verifying Fixes

Documented Buggy Version	Incremental Analysis				Multiversion Analysis					
	Detected Bugs	T(s)	Fixed	T(s)	Releases	Doc	Impacted	T(s)	Fixed	T(s)
gzip-1.2.4	Buffer Overflow	0.12	Yes	0.08	4	1	4	0.13	4	0.69
libpng-1.5.14	Integer Overflow	0.28	No	0.24	6	2	6	0.48	0	1.45
libpng-1.5.14	Integer Overflow	0.24	Yes	0.18	6	2	6	1.45	5	1.22
tightvnc-1.3.9	Integer Signedness	2.6	Yes	0	4	1	4	4.8	4	0
putty-0.55	Null-Pointer Deref	20.0	Yes	0.07	3	1	1	26.1	1	0.09

construct an MVICFG consisting of the buggy version, shown under *Documented Buggy Version*, its corresponding patched version and a set of releases. The documentation provides the location of the bug in the source code and also the types of the bug. Under *Detected Bugs*, we list a set of documented bugs confirmed by Hydrogen. Under *Fixed*, we show whether the patches correctly fixed the bugs. Under *T(s)*, we report the time in seconds used to detect the bugs and verify the fixes. Our results show that using incremental analysis, Hydrogen can quickly verify the fixes for a set of real-life bugs, including buffer overflows, integer overflows, integer signedness conversion problems and null-pointer dereferences. The integer overflows in *libpng* shown in the 2nd and 3rd rows in the table are the same bug located in different revisions, reported from *libpng-1.5.14*. We correctly identified that this integer overflow is not correctly fixed by the first patch (see row 2) and then correctly fixed by the second patch (see row 3).

We report our analysis results for multiple versions of programs under *Multiversion Analysis*. Under *Total V*, we list the number of releases integrated on the MVICFG. Under *Doc* and *Impacted*, we compare the documented results with the results Hydrogen reports regarding how many of such releases are impacted by the bug. Our data show that Hydrogen detected more releases impacted by the bug than the documentation says. After manually inspecting the results, we find that Hydrogen correctly reported all the impacted versions, and the documentation is incomplete. The manual inspection finds that the second patch for the integer overflow of *libpng* successfully fixed all the 6 versions, and we reported 5, shown in the 3rd row. The imprecision is caused by the fact that Phoenix does not provide the information for a structure member needed for correctness. We run this experiments on a Windows machine with 4 duo Intel Core i7-2600 CPU and 16.0 GB memory. Columns *T(s)* show that our analysis is very efficient and reports the results in seconds for large programs and for multiple versions.

An interesting case we found in our experiments is that *libpng 1.6.0* was released after the patch for *libpng 1.5.14* was developed. We would have assumed that the *libpng 1.6.0* was already patched; however, Hydrogen reports that the bug still exists in *libpng 1.6.0*. We found in the code comments, developers have written *TODO: fix the potential overflow*. This indicates that manual patch management can be error-prone, and we need tools such as Hydrogen to provide automatic support for tasks of determining the bug impact and propagating the patches across branches.

6.2.3 Detecting Bugs in Changes

In this experiment, we randomly selected two versions of the benchmark programs, shown under *Benchmark Versions*, and performed change-based and cache-based analyses on detecting bugs for the changes between the two versions. We focused on the three types of bugs, buffer overflows, integer overflows and null-pointer dereferences. In Table 3, under *Total PFS*, we list the number of PFSs reported for the three types for analyzing the entire version of the program. Compared to the PFSs detected in the *cache-based*

analysis and *change-based* analysis shown under *PFS*, the data suggest that we can significantly benefit from program analysis targeting changes for verifying a new version. In addition to *PFS*, we identified the other three metrics for comparing efficiency. Under *V-B* and *V-P*, we report the number of blocks and procedures visited during the analysis, and under *T(s)*, we give the time used to run the analysis. Under *W*, we list the number of warnings reported.

Our experimental results show that we can detect bugs in changes in seconds for most of the benchmarks using both of the analyses. The efficiency of the analysis provides promising evidences that we can potentially deploy such analysis at the code check-in time to verify changes and provide developers immediate feedback. The results further show that cache-based analysis is more efficient than change-based analysis. For *putty*, the change-based analysis runs out of memory, and we thus are not able to obtain the bug detection results. However, in the cache-based analysis, we still verified all the PFSs in changes for bugs, although our analysis for computing the cached queries is terminated before traversing all the paths, and we may miss bugs related to changes.

The experimental results also demonstrate that the cache-based analysis reports fewer warnings and provides more focuses on confirming and diagnosing the bugs. We manually confirmed the warnings generated from the cache-based analysis. We found that the 4 warnings we identified for *libpng* are all real integer overflows. In fact, 2 are related to a vulnerability reported (we did not know the bug before detecting it) and one are related to the memory allocation size that potentially can cause a buffer overflow. The 2 buffer overflow reported for *putty* are false positives due to the lack of precise pointer analysis.

6.3 Summary and Discussions

In this section, we provided experimental evidences to show that the construction of an MVICFG is scalable (Table 1), correct and useful (Tables 2 and 3). We have used a set of real-life programs, bugs and patches to demonstrate that we are able to automatically perform patch verification and identify the information that is missing in the documentation. We are also able to efficiently find bugs in the changes compared to other approaches.

Intuitively, an MVICFG is a compact representation that specifies the control flow differences and commonalities for a set of programs. When the programs integrated on the MVICFG share the majority of code, the analysis can benefit most by reusing the intermediate analysis results across versions. The MVICFG represents fine-grain, semantic comparisons such as control flow differences within a function; thus it effectively characterizes the behavioral differences for program versions that contain many small changes in-between. Meanwhile, the coarse grain changes such as adding or removing functions are also available on the MVICFG, and we can perform analysis to determine properties for such changes as well, e.g., which calls can be impacted by the newly added function.

Through the experimentation, we have also found a set of potential improvements we can make for the MVICFG. First, in our

Table 3: Detecting Bugs in Changes

Benchmark Versions	Total PFS	Cache-based, Demand-Driven Analysis					Change-based, Demand-Driven Analysis				
		PFS	V-B	V-P	T(s)	W	PFS	V-B	V-P	T(s)	W
tcas, 1-2	21	3	36	8	0.17	0	3	36	8	0.37	0
schedule, 1-2	68	1	33	12	0.67	0	8	983	376	10.9	0
printtokens, 1-2	56	3	3	3	0.005	0	6	63	6	0.85	0
gzip, 1.1.2-1.2.2	184	15	21	15	0.08	0	160	5.0 k	201	26.9	16
tightvnc, 1.2.0-1.2.2	1449	1	1	1	0.09	0	9	9	9	0.8	0
libpng, 1.0.57-1.2.48	3187	8	1.9 k	1.7 k	1.87	4	316	76.2 k	48.7 k	352.8	42
putty, 0.53-0.55	8127	19	23.9 k	503	187.1	2	1587	-	-	-	-

current approach of building an MVICFG, we integrate one version a time and compare the statements between versions to determine the changes. Although the MVICFG built can satisfy the requirements (see Section 3) and correctly perform the patch verification shown above, the change specified on the MVICFG may not always reflect programmers’ intention on what is the change. Consider in version 2, we delete a statement, and then we add the statement back in version 3. On the current MVICFG, we would integrate a deletion between versions 2 and 1 and then an addition between versions 3 and 2. The match between any non-consecutive versions is not directly shown on the graph unless we compare the program with all the other versions rather than just with its previous version. In addition, analyzing the MVICFG, we can further improve the precision by performing an alias analysis on the MVICFG.

7. RELATED WORK

In the keynote at the PASTE workshop in 2002, Notkin proposed the concept of longitudinal program analysis and the need for reuse and learning information retained from earlier analysis to a new version of software [37]. The MVICFG is a program representation to enable such analyses. The MVICFG is related to techniques on representing program differences, analyzing program changes and software history and also detecting bugs in the programs. In the following, we present the comparisons of these areas and our work.

Program Differencing. Previous techniques on comparing programs mainly focus on two versions [4, 21, 29, 30, 43, 56, 19]. The MVICFGs enable the comparisons for multiple versions and characterize behavioral differences. We highlight three representative solutions closely related to the MVICFG. Horwitz et al. performed code differencing on dependency graphs to determine noninterference changes [23, 24]. Dependency graphs are adequate to determine whether a change impact exists but not able to provide further reasoning on the changes of program properties as we have done. Raghavan et al. applied graph differencing algorithms on *abstract semantic graphs (ASG)* [44]. The ASG is an abstract syntax tree annotated with semantic information. The differences available via ASGs do not include the comparisons on important semantics such as program paths. Apiwattanapong et al. performed the comparisons on control flow graphs using Hammock based approaches [5]. The differences are marked on the individual control flow graphs rather than displaying in a union representation like MVICFGs that can enable program analyses.

Program Analysis and Testing on Changes. A foundation of change-based program analysis is *impact analysis*, a type of dependency analysis that determines the propagation of change effects for solving problems such as regression testing [7, 17, 31, 38, 42, 45, 55]. Here, we compare the two most relevant change-based program analyses. Person et al. conducted program differencing on symbolic execution trees and applied it for selecting regression

test cases [39, 40]. The comparison exhaustively collects symbolic signatures and has not shown scalability on interprocedural analysis. Yang et al. used impact analysis to isolate the code potentially affected by the changes and performed model checking only on the impacted code [55]. On the MVICFGs, the changed paths can be identified via a reachability analysis, and we can easily cache and reuse the analysis results from previous versions. Based on a demand-driven algorithm, our incremental analysis is believed to be more scalable and flexible than the two above approaches.

Multiple Versions and Software History. Research interests on program languages and analysis for multiple versions and software history are mostly recent [18, 47, 48, 57]. Erwig et al. developed *choice calculus*, a language for manually specifying software variants with the goal of better developing changes. Our approach does not involve manual effort for specifying the differences. Servant et al. proposed *history slicing*, showing how the same code locations evolve in terms of definition and use of the variables. We are able to identify such information with Hydrogen and expect to be more precise with the path-sensitive, demand-driven symbolic analysis on MVICFGs.

Static Bug Detection. Due to its importance, there has been much work on bug detection and diagnosis [12, 20, 33, 34, 35, 46]. Among the techniques, demand-driven has shown scalability [33, 34, 35], and path-sensitive analysis has the advantages of being precise and able to provide rich information [6, 16, 33, 35, 54, 53].

8. CONCLUSIONS

This paper presents a program representation, the MVICFG, that specifies software changes and compares program versions using program control flow, and a demand-driven, path-sensitive, symbolic analysis on the MVICFG that determines the commonalities and differences of program properties for a set of program versions. The key impact of an MVICFG is twofold. First, it enables reuse of the program analysis results from previous versions for scalable, yet still precise, program verification to solve software assurance problems related to incremental software development process. Second, it makes possible not only analyze programs along their execution paths but also longitudinally across program versions for efficient online comparisons of advanced program properties. We demonstrate the usefulness and practicability of the framework for a set of important problems related to patch verification. In the future, we will further explore other types of program analyses on MVICFGs to determine program properties related to software changes and program versions.

9. ACKNOWLEDGMENTS

We thank Suzette Person for early discussions of the work.

10. REFERENCES

- [1] Personal communication with developers at Google, 2011.
- [2] Personal communication with developers at Firefox, 2013.
- [3] M. Acharya and B. Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 746–755, New York, NY, USA, 2011. ACM.
- [4] T. Apiwattanapong, A. Orso, and M. J. Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engg.*, 14:3–36, March 2007.
- [5] T. Apiwattanapong, A. Orso, and M. J. Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engg.*, 14(1):3–36, Mar. 2007.
- [6] D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In *ICSE'08: Proceedings of the 30th international conference on Software engineering*, 2008.
- [7] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Software Maintenance, 1992. Proceedings., Conference on*, pages 41–50, nov 1992.
- [8] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *FSE'05: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 1997.
- [9] Buggy Apps Are not Popular. <http://techcrunch.com/>, 2013.
- [10] Bugzilla. <http://www.bugzilla.org/>, 2012.
- [11] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 2000.
- [12] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12:10:1–10:38, December 2008.
- [13] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *CCS'02: Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.
- [14] Common Vulnerability Exposure. <http://cve.mitre.org/>, 2013.
- [15] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI'02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.
- [16] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 57–68, New York, NY, USA, 2002. ACM.
- [17] S. Elbaum, D. Gable, and G. Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Software Metrics Symposium, 2001. METRICS 2001. Proceedings. Seventh International*, pages 169–179, 2001.
- [18] M. Erwig. A language for software variation research. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 3–12, New York, NY, USA, 2010. ACM.
- [19] B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. In *IEEE Trans. on Software Engineering*, volume 33, pages 725–743, 2007.
- [20] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.
- [21] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 55–64, New York, NY, USA, 2010. ACM.
- [22] Y. Hamadi. Disolver : A Distributed Constraint Solver. Technical Report MSR-TR-2003-91, Microsoft Research, 2002.
- [23] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 234–245, New York, NY, USA, 1990. ACM.
- [24] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.
- [25] J. Humble and D. Farley. *Continuous Delivery*. Addison Wesley, 2010.
- [26] J. W. Hunt and M. D. McIlroy. An Algorithm for Differential File Comparison. Technical report, Bell Laboratories, 1976.
- [27] Ian Kaplan. A semantic graph query language. UCRL-TR-225447, 2008.
- [28] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams. Do faster releases improve software quality? an empirical case study of mozilla firefox. In *MSR*, pages 179–188. IEEE, 2012.
- [29] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 309–319, 2009.
- [30] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *Software Maintenance, 1992. Proceedings., Conference on*, pages 282–290, nov 1992.
- [31] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 308–318, may 2003.
- [32] W. Le. Segmented symbolic analysis. In *Proceedings of the International Conference on Software Engineering, SIGSOFT ICSE 2013*, 2013.
- [33] W. Le and M. L. Soffa. Marple: a demand-driven path-sensitive buffer overflow detector. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 272–282, New York, NY, USA, 2008. ACM.
- [34] W. Le and M. L. Soffa. Path-based fault correlations. In *FSE'10: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2010.
- [35] W. Le and M. L. Soffa. Generating analyses for detecting faults in path segments. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 320–330, New York, NY, USA, 2011. ACM.
- [36] Mozilla Crash Reporter. <https://support.mozilla.org/en-US/kb/MozillaCrashReporter>, 2013.

- [37] D. Notkin. Longitudinal program analysis. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '02, pages 1–1, New York, NY, USA, 2002. ACM.
- [38] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 491–500, Washington, DC, USA, 2004. IEEE Computer Society.
- [39] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237, Atlanta, Georgia, 2008. ACM.
- [40] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 504–515, New York, NY, USA, 2011. ACM.
- [41] Phoenix. <http://research.microsoft.com/phoenix/>, 2004.
- [42] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: an approach for debugging evolving programs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 33–42, New York, NY, USA, 2009. ACM.
- [43] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 188–197, Washington, DC, USA, 2004. IEEE Computer Society.
- [44] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 188–197, Washington, DC, USA, 2004. IEEE Computer Society.
- [45] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Trans. Softw. Eng. Methodol.*, 13:277–331, July 2004.
- [46] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for c. In *FSE'05: Proceedings of the 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005.
- [47] F. Servant and J. A. Jones. History slicing: assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 43:1–43:11, New York, NY, USA, 2012. ACM.
- [48] V. S. Sinha, S. Sinha, and S. Rao. BUGINNINGS: identifying the origins of a bug. In *Proceedings of the 3rd India software engineering conference*, pages 3–12, Mysore, India, 2010. ACM.
- [49] SIR benchmark. <http://sir.unl.edu/portal/index.php>, 2004.
- [50] srcML. <http://www.sdml.info/projects/srcml/>, 2012.
- [51] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 51:1–51:11, New York, NY, USA, 2012. ACM.
- [52] UNIX Diff. <http://unixhelp.ed.ac.uk/CGI/man-cgi?diff>.
- [53] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29, May 2007.
- [54] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *FSE'03: Proceedings of 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003.
- [55] G. Yang, M. Dwyer, and G. Rothermel. Regression model checking. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 115–124, sept. 2009.
- [56] W. Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21:739–755, June 1991.
- [57] J. Yi, D. Qi, and A. Roychoudhury. Expressing and checking intended changes via software change contracts. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '13, 2013.
- [58] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 26–36, New York, NY, USA, 2011. ACM.