Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2023-IC-008

2023-IC-008

Incremental Call Graph Construction in Industrial Practice

Zelin Zhao, Xizao Wang, Zhaogui Xu, Zhenhao Tang, Yongchao Li, Peng Di

Technical Report 2023

Incremental Call Graph Construction in Industrial Practice

Zelin Zhao*, Xizao Wang*†, Zhaogui Xu*, Zhenhao Tang*, Yongchao Li*, Peng Di*‡
Ant Group, Hangzhou, China* *Nanjing University*, Nanjing, China†
{zelin.zzl, wangxizao.wxz, zhengrong.xzg, tangzhenhao.tzh, liyongchao.lyc, dipeng.dp}@antgroup.com

***Abstract*—Interprocedural program analysis is critical in finding hidden program defects and vulnerabilities in CI/CD pipelines. A pre-constructed call graph is a prerequisite for interprocedural analysis. However, the exhaustive call graph construction, i.e., analyzing the target program as a whole and constructing from scratch often takes too much time. We made a scalable empirical study on both industrial and open-source projects and observed that most program updates only involve a very limited part of the code. The observation inspires an efficient approach that not wholely re-constructs a call graph but incrementally patches the old one with the partial graph affected by the update. We propose a sound incremental call graph construction algorithm that works in a reset-recompute way: first, prune invalid nodes and edges from the old call graph, then analyze the new code to patch it to construct the new one. We implemented the algorithm and built a benchmark suite consisting of 20 industrial and 10 open-source projects. The experimental evaluation shows that the efficiency improvement is encouraging. Compared with the exhaustive construction algorithm, the incremental way can speed up the construction by 20.0 times and reduce the memory and storage consumption to 58.1% and 10.4%, respectively.**

***Index Terms*—call graph, incremental construction, class hierarchy analysis, CI/CD**

I. Introduction

Continuous integration and delivery (CI/CD) pipelines are now widely practiced by modern software enterprises to manage program development. Interprocedural program analysis is a promising step in the CI/CD pipeline for finding program defects and vulnerabilities. Broadly, interprocedural analyses are classified into top-down and bottom-up [1], both kinds of which require a call graph (CG) to propagate information between callers and callees. The whole CI/CD pipeline should finish in a few minutes in agile practice [2], [3]. Therefore, the efficiency of constructing CG as a fundamental program representation has become a key metric to CI/CD pipelines.

The ever-growing program size and agile development of industrial software pose great challenges for effectively constructing call graphs. Most existing CG construction algorithms are designed in an *exhaustive* style, *i.e.*, analyze the target program as a whole and construct CG from scratch. For complicated programs, these algorithms require too much time to meet the requirement of CI/CD. The widely-used *class hierarchy analysis* (CHA) algorithm [4] takes over 9 minutes for industrial programs (with 367K LOC on average in our evaluation) to construct CG (see section V), which exceeds the suggested time limit [5]–[7]. Frequent program updates make the situation even worse in CI/CD. An Ant Group application releases 294 new versions in only one year (see section V). On the one hand, the successive updates of a program require much storage space to save different versions of full CG; on the other hand, the simultaneous releases of different programs require much computing resources and time to construct CG. Moreover, no matter how much code changes in the update, the old CG becomes stale, and the new CG must be constructed from scratch in existing practices.

Patching the old CG to construct the new one, just like patching the code, is an ideal way of CG construction in the CI/CD pipeline. According to our observation, most program releases only update a small part of the code and leave the major part unchanged. 40.93% of updates in Ant Group only change less than 100 LOC (lines of code), and 72.45% of updates change less than 1,000 LOC. The CG patch to these updates could be small-scale. The observation inspires an efficient approach which not wholely re-constructs CG but incrementally patches the previous CG with the partial graph affected by the updates. However, the incremental CG construction is not trivial, because updates may explicitly and implicitly affect the call graph. Therefore, all impacts are supposed to be captured during construction to guarantee the new CG is valid and sound, and the consumption of resources and time has to meet the limits.

In this paper, we propose an incremental CHA-based CG construction algorithm for industrial Java applications. The approach first collects all the method-level program updates as inputs, capturing the explicit effects to CG; then locates unchanged methods that are affected by the updates, capturing the implicit effects. The incremental construction works in a *reset-recompute* way: first, prune invalid nodes and edges from the old CG, then analyze the new code to patch the old CG to construct the new one. Existing incremental CG construction approach can not be easily adopted into CI/CD infrastructures, because they are designed for specific IDE [8] or special programs [9], or require too much memory [10].

The key advantage of our incremental algorithm over the existing one [4] is that reusing the unchanged part of the old CG, which is the major part for most program updates, significantly saves computing resources and time. Compared with the full CG construction, the patch, consisting of the removed and added nodes and edges, is a tiny scale and takes much less storage space. Our algorithm is theoretically as

‡ Corresponding author.

2832-7659/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE-SEIP58684.2023.00048

sound as the full CHA-based algorithm. The new CG consists of two sound parts: the reused part is constructed soundly for the code irrelevant to the update, and the new part is constructed soundly for the code relevant to the update.

Our contributions are summarized as follows.

- We conduct an empirical study of 9,059 updates (releases) from 108 industrial and 20 open-source programs. The results confirm our observation that most program updates only involve a small part of the code.
- We propose a novel incremental CG construction algorithm, which can capture all impacts caused by the program update and construct a valid and sound new CG.
- We implement the proposed algorithm and evaluate it by comparing it with the full CG construction on 735 updates from 30 projects (with 20 real industrial and 10 open-sourced projects). On average, the incremental CG construction algorithm speedups the construction by 20.0 times and reduces the memory and storage consumption to 58.1% and 10.4%, respectively.

The rest is organized as follows. Section II gives the background and motivation with an example and discusses the challenges of incremental CG construction. Section III introduces the original CG algorithm, and section IV presents our incremental CG algorithm. The evaluation is shown in section V, followed by the discussion of threats in section VI, the overview of related work in section VII, and the conclusion in section VIII.

II. Motivation and Challenge

In this section, we first present the motivation for incremental CG construction, then discuss the challenges of conducting this idea.

A. Motivation

Most CG construction algorithms are designed in an exhaustive style, *i.e.*, analyzing the whole program to construct the full CG from scratch. Such a design costs a lot, especially for large-scale industrial programs. Our later experiment shows that the CHA [4] algorithm can take over 22 minutes and 18 GB of memory to construct the CG for an industrial program (with 339K LOC). Moreover, an industrial program usually evolves continually in the CI/CD pipeline. Constructing a full CG for each program update increases the resource consumption of these full construction algorithms.

The key idea is to patch the old CG to incrementally construct a new CG, which is feasible for real program updates. Our insight is that the update from the old CG to the new CG must be correlative to the updated code, which should be located and analyzed. It is also observed that a program update usually only modifies a small part of the code, and the resultant CG update is likely small-scale.

Figure 1 shows a simplified update of an industrial program. There are 6 classes, 7 methods, and 3 fields in the new code related to the update, while a considerable amount of unchanged code is not shown. Existing exhaustive CG construction algorithms (*e.g.*, CHA [4]) must analyze the new

```
1   class UserService {
2     private List<User> users;
3     public void add(User user) { ... }
4   }
5   class VipService extends UserService {
6+    private VipVerifier vipVerf;
7     @Override
8     public void add(User user) {
9+      if (vipVerf.isVip(user))
10      super.add(user);
11    }
12  }
13- class BlackListService extends UserService {
14+ class BlockedService extends UserService {
15    @Override
16    public void add(User user) { ... }
17  }
18- class FooService extends UserService {
19+ class FooService extends Foo {
20    @Override
21    public void add(User user) { ... }
22  }
23  class Server {
24    private UserService service;
25    public void init(UserService userService) {
26      this.service = userService;
27    }
28    public void userRegister(User user) {
29      service.add(user);
30    }
31  }
32  class VipVerifier {
33    public boolean isVip(User user){ ... }
34  }
```

Fig. 1: Example of a program update. Red lines are removed, green lines are added, and others are unchanged.

code as a whole, including the unchanged code. By patching the old CG, however, we can ignore almost all unchanged code and only analyze the 3 changed classes and 1 class (`Server`) that is unchanged but affected by the update, *i.e.*, the *virtual call* resolution in line 29 becomes different.

To confirm our observation, we conducted an empirical study of the program evolution history to measure the scale of updated code in the real world. We build a benchmark suite containing both industrial and open-source programs. For industrial programs, we randomly chose 108 core applications in Ant Group and collected their updates (releases) from April 2021 to April 2022, and got 7,262 updates. For open-source programs, we chose 20 widely used Maven [11] projects which have different purposes (libraries, cli tools, and gui tools), collected all updates, and got 1,797 updates.

Figure 2 shows the results of the empirical study, where each update is measured with its updated LOC. The updates with less than 100 updated LOC occupy a big part, and the ones with less than 1,000 updated LOC occupy the major part. For such simple updates, incremental CG construction may only need to update a small part of the old CG, which is much cheaper than the original way. A small part of updates are complex and changes more than 1,000 or even 10,000 LOC. Incremental CG construction for a complex update may take a longer time than that for a simple one.

The insight and empirical observation motivate us to reform

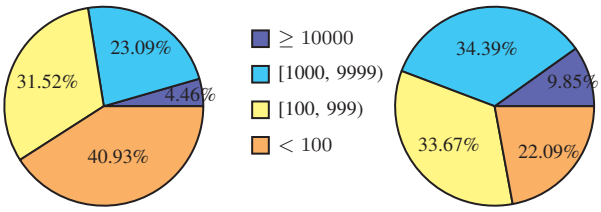


Fig. 2: Ths statistics of updated LOC in the evolution history of programs. The left pie is for industrial programs, and the right is for open-source programs.

the CHA algorithm to construct a CG for a program update incrementally. The CHA algorithm is widely used and suitable for CI/CD. On the other hand, it is the basis of other algorithms, such as RTA [12], VTA [13], *etc*. The incrementalization of CHA is the precondition of other incremental construction algorithms. We will discuss this in section VII.

B. Challenges of Incremental CG Construction

1) Capture Different Update Effects on CG: Program updates can *explicitly* and *implicitly* affect the CG. These update effects on CG should all be captured by incremental CG construction.

The explicit effect means that, a method invocation is updated, and its CG edges must be updated accordingly. For example, line 9 in fig. 1 adds a new method invocation and introduces an explicit effect to CG, *i.e*., adding a new CG edge ⟨`VipService#add`,`VipVerifier#isVip`⟩.

The implicit effect means that, although a method invocation is unchanged, its CG edges should be updated due to the program update. For example, lines 13–14 in fig. 1 first remove and then add a subtype for `UserService`. The sub methods of `UserService#add` become different. So that the CG edges of the unchanged virtual call in line 29 become different.

2) Construct a Valid and Sound CG: The CHA algorithm exhaustively analyzes a program and ensures the *validity* and *soundness* of the class hierarchy and call graph. The incremental construction must also preserve these properties.

Validity: Each node and edge in the class hierarchy and call graph must be valid in the corresponding version of a program. Because of program updates, types and inheritances, methods and invocations can be removed from program code. The corresponding nodes and edges become invalid and will be removed from the new class hierarchy and call graph.

Soundness: A sound class hierarchy contains all (in)direct super types for each type, thus all possible callees in the call graph can be resolved via a sound class hierarchy. Because of program updates, types and inheritances, methods and invocations can be added to the new version of program. The corresponding nodes and edges must be added to the new class hierarchy and call graph.

The updating class hierarchy and call graph sequence also affect their validity and soundness. Suppose updating the call graph before the class hierarchy, invalid inheritance edges may still exist in the class hierarchy (such as ⟨`FooService`,`UserService`⟩ in fig. 1), and the virtual call resolution can generate invalid CG edges (*i.e*., ⟨`UserService#userRegister`,29,`FooService#add`⟩ is invalid in the new program). Suppose updating the class hierarchy before the call graph, invalid invocation nodes (such as `BlackListService#add`) may still exist in the call graph and introduce invalid types (*i.e*., `BlackListService`).

III. Full Call Graph Construction

The class hierarchy is a graph that describes the inheritance between classes in a program. In this graph, nodes are classes in a program, and edges are inheritance relationships pointing from each class to its direct super types. In this paper, we use ⟨`superType`,`subType`⟩ to denote a class hierarchy edge and ⟨`caller`,`call-site`,`callee`⟩ [1] to denote a CG edge. Take the old code in fig. 1 as an example. The class hierarchy contains the following edges:

- ⟨`VipService`,`UserService`⟩,
- ⟨`BlackListService`,`UserService`⟩,
- ⟨`FooService`,`UserService`⟩.

The call graph contains the following edges for line 29:

- ⟨`Server#userReg`,29,`UserService#add`⟩,
- ⟨`Server#userReg`,29,`VipService#add`⟩,
- ⟨`Server#userReg`,29,`BlackListService#add`⟩,
- ⟨`Server#userReg`,29,`FooService#add`⟩.

A. CHA-based CG Construction Algorithm

The CHA algorithm assumes a class hierarchy is already available before CG construction [4]. In this paper, we present the CHA algorithm more practically, *i.e*., building the class hierarchy and call graph in one run.

The CHA algorithm is shown in alg. 1. It takes as input the entry methods of the target program, usually the `main` method, and constructs the class hierarchy and the call graph from scratch. Line 2 initializes the class hierarchy $\mathbb{H}$ and the call graph $\mathbb{G}$ as two empty graphs. Lines 3–9 iteratively update the two graphs until they do not change, *i.e*., reaching the fixed point. Before each iteration, M is initialized as all methods from input (M^e) and $\mathbb{G}$ in line 4. The main logic in lines 5–8 is visiting each method to first update $\mathbb{H}$ (line 7) and then update $\mathbb{G}$ (line 8).

Line 7 invokes `createCH` to create a partial class hierarchy to update $\mathbb{H}$. Lines 13–14 iterate through types directly used by the method and collect their (in)direct super types. Line 15 creates direct inheritance edges for each collected type. These edges are added to the partial class hierarchy $\mathbb{H}'$.

Line 8 invokes `createCG` to create a partial call graph to update $\mathbb{G}$. It first initializes an empty call graph, then iterates through each method invocation in this method. The class hierarchy is necessary for resolving virtual calls. Therefore $\mathbb{H}$ is a parameter of the `createCG` method. The `resolveMethods` resolves all possible callees via $\mathbb{H}$ for each invocation. Line 21 creates CG edges and adds them to

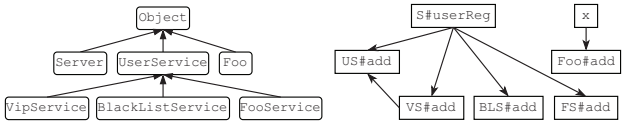
[1]The caller can invoke a callee multiple times in different call-sites. For simplicity, we denote a call-site by its line number in this paper.

Algorithm 1: Full CHA-based CG algorithm

```
   Input: M^e, the entry methods
   Output: H, the full class hierarchy;
           G, the full call graph
1  Function fullChaBasedCG(M^e):
2      H ← {}, G ← {};
3      repeat
           // visit entry methods or reachable ones
4          M ← M^e ∪ nodes(G);
5          foreach m ∈ M do
6              if m was not visited before then
7                  H ← H ∪ createCH(m);
8              G ← G ∪ createCG(H, m);
9      until H and G do not change;
10     return H, G
11 Function createCH(m):
12     T ← {};
13     foreach type t directly used by m do
           // collect t and its (in)direct super types
14         T ← T ∪ {t} ∪ allSuperTypes(t);
       // create a partial class hierarchy
15     H' ← {⟨t, t'⟩ | t ∈ T ∧ t' ∈ directSuperTypes(t)};
16     return H'
17 Function createCG(H, m):
18     G' ← {};
19     foreach method invocation i in m do
           // resolve possible callees for i using H
20         M ← resolveMethods(H, i);
21         G' ← G' ∪ {⟨m, i, m'⟩ | m' ∈ M};
22     return G'
```

We simplify graph denotation to the set of edges, and use nodes() to retrieve methods from the call graph.

$\mathbb{G}'$. Each newly reachable method will be finally added to M in line 4 and visited in the next iteration. The class hierarchy and the call graph grow successively until they reach the fixed point.



(a) Class hierarchy (b) Call graph

Fig. 3: The class hierarchy and call graph constructed by the CHA algorithm for the old code in fig. 1. Assume that the old code does not use `VipVerifier`, and that method `x` calls `Foo#add`. Class names in CG are abbreviated to save space.

Figure 3 presents the two graphs constructed for the old code in fig. 1 by the CHA algorithm. We assume `VipVerifier` is not reachable in the old code. Therefore the two graphs do not contain its type and methods. While analyzing the `UserService#add` method, the inheritance edge from `UserService` to `Object` is created. The resolution for line 29 in `userRegister` can be sound only after the three subclasses of `UserService` are added to the class hierarchy.

IV. Incremental Call Graph Construction

Alg. 2 presents our incremental CHA-based CG algorithm, IncCHA. The inputs contain a program update Δ, the old class hierarchy $\mathbb{H}_o$, and the old call graph $\mathbb{G}_o$. IncCHA mainly contains six steps.

Algorithm 2: Incremental CHA-based CG algorithm (IncCHA)

```
   Input: Δ, a program update;
          H_o, the old class hierarchy;
          G_o, the old call graph
   Output: H_n, the new class hierarchy;
           G_n, the new call graph
1  Function incrementalChaBasedCG(Δ):
       // Step 1: obtain all updated methods
2      M_Δ ← processUpdate(Δ);
3      initialize H_n and G_n to duplicate of H_o and G_o;
       // Step 2: prune the class hierarchy
4      foreach m ∈ nodes(G_o) ∩ M_Δ do
5          H_n ← H_n − createCH(m);
6      repeat
           // Step 3: collect updated and affected methods
           // collect newly reachable methods
7          M_r ← nodes(G_n) − nodes(G_o);
           // collect super methods in the old version
8          M_o ← oldSuperMethods(M_Δ ∪ M_r);
           // collect super methods in the new version
9          M_n ← newSuperMethods(M_Δ ∪ M_r);
           // collect direct callers in the old call graph
10         M_c ← {m | m' ∈ M_Δ∪M_o∪M_n ∧ ⟨m, i, m'⟩ ∈ G_o};
           // Step 4: prune the call graph
11         foreach m ∈ M_Δ ∪ M_c do
12             if m was not visited before then
13                 G_Δ ← {⟨m, i, m'⟩ | ⟨m, i, m'⟩ ∈ G_o};
14                 G_n ← G_n − G_Δ;
           // Step 5: analyze new method to update H_n and G_n
15         foreach m ∈ (M_Δ ∪ M_c ∪ M_r) ∩ nodes(G_n) do
16             if m was not visited before then
17                 H_n ← H_n ∪ createCH(m);
18             G_n ← G_n ∪ createCG(H_n, m);
19     until H_n and G_n do not change;
       // Step 6: clean non-entry nodes whose in-degree is 0
       cleanUp(G_n);
```

A. Incremental Construction Algorithm

Step 1: Obtain updated methods. For different element updates in Δ, `processUpdate` parses them to obtain updated methods and adds them to M_Δ, as shown in table I. A class that only exists in one version of the program is an added or removed class, and all its declared methods are added or removed methods correspondingly. For the two versions of a changed class, their declared methods are first matched by their signatures[2]. Unmatched methods are removed or added methods. For matched methods, their method bodies are then compared to distinguish unchanged from changed methods. If the fields, import statements, or direct super types of a class are changed, like `FooService` in fig. 1, its methods are treated as changed methods, except those removed and added ones.

The M_Δ column in table II presents the updated methods obtained by this step. The order of these methods does not affect incremental construction.

Step 2: Prune the Class Hierarchy. To ensure the validity of $\mathbb{H}_n$ and $\mathbb{G}_n$ (section II-B2), IncCHA first finds and removes invalid types and inheritances via lines 4–5. After this step,

[2]Method signature consists of the declaring class name, method name, parameter types, and return type.

TABLE I: The updated methods added to M_Δ, by processing different updates

Update Category	Methods Added to M_Δ
Add class	set of added methods, {+m}
Remove class	set of removed methods, {−m}
Add method	an added method, +m
Remove method	a removed method, −m
Change method	a changed method, ⋆m
Update fields, imports, and super types of class *etc.*	a set of changed methods, {⋆m}

"+", "−" and "⋆" denote added, removed and changed.

all nodes and edges in $\mathbb{H}_n$ are valid for the new program. Therefore the subsequent virtual call resolutions will be valid. Invalid CG edges are removed in line 14, which will be described in line 20.

For removed and changed methods in M_Δ that are reachable in the old program, line 5 reuses the `createCH` function from the full CHA algorithm to build the partial class hierarchy. In fig. 1, the `add` method in `FooService` is processed as a changed method. When `createCH` parsing the `add` method in the old program, the type inheritance edge, ⟨`FooService`,`UserService`⟩, is captured in the partial class hierarchy and finally removed from $\mathbb{H}_n$ in line 5.

Valid type inheritance edges may also be captured by `createCH`, such as ⟨`VipService`,`UserService`⟩ when analyzing `VipService#add` method. These edges will be added back into $\mathbb{H}_n$ in line 17 when visiting the corresponding new version of methods. Moreover, we can optimize such superfluous deletions by skipping valid type inheritance edges.

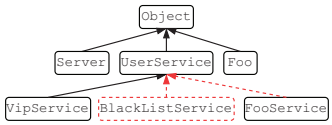


Fig. 4: Removing invalid types and inheritance edges (red dashed ones) from $\mathbb{H}_n$.

Figure 4 shows the new class hierarchy after this step. The remaining types and edges ensure the validity of virtual call resolution. IncCHA can directly use $\mathbb{H}_n$ to resolve virtual calls in subsequent construction. While resolving line 29 in fig. 1, IncCHA will not generate the invalid edge pointing to `FooService#add`.

Step 3: Collect Updated and Affected Methods. The outgoing edges from updated methods and the incoming edges to updated methods should be recreated. IncCHA (lines 7–10) finds all methods that are affected by the program update.

Methods that are unreachable in the old program but become reachable in the new program should be analyzed. In line 7, the `nodes(`$\mathbb{G}_n$`)−nodes(`$\mathbb{G}_o$`)` operation computes such methods. These methods could be the added ones, or the existing ones in both versions but are never called in the old program. CG edges originating from them should be created.

Callers that could invoke updated or newly reachable methods should be analyzed. Besides the invocations directly pointing to these methods, invocations that point to their super methods should also be found. IncCHA (lines 8–9) finds all super methods M_o and M_n for updated and newly reachable ones in the old and the new program. Line 10 walks on the old call graph to obtain direct callers. If there is no old bytecode, the `oldSuperMethods` computes the super methods based on the old class hierarchy and the old call graph.

TABLE II: Methods added into variables of lines 7–10 in IncCHA

M_Δ	M_r	$M_o \cup M_n$	M_c
① ⋆VS#add ② −BLS#add ③ +BS#add ④ ⋆FS#add		US#add	⑤ S#userReg
① ⋆VS#add ② −BLS#add ③ +BS#add ④ ⋆FS#add	⑤ VV#isVip	US#add Foo#add	⑥ S#userReg ⑦ x

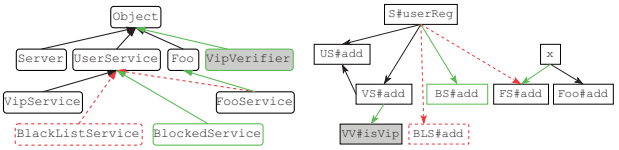
The first and second row shows the states in the first and second iteration, respectively. We put M_o and M_n in one column for saving space. Numbered methods are finally visited in each iteration.

Table II presents the states of variables in lines 7–10. In the first iteration (*i.e.*, first row), M_Δ contains the four updated methods. Because the initial $\mathbb{G}_n$ is the duplicate of $\mathbb{G}_o$, there are no newly reachable methods in M_r. When collecting super methods, the new inheritance edge from `FooService` to `Foo` has not been created, so there is only one super method in M_o. The direct caller in the first iteration is only `userRegister` in `Servce` class.

In the second iteration (*i.e.*, second row), M_Δ remains unchanged. Because the new version of `VipService#add` is visited in the previous iteration, `VipVerifier#add` becomes newly reachable in the new program. Because the new version of `FooService#add` is visited in the first iteration, the new inheritance edge from `FooService` to `Foo` is created, and the new super method `Foo#add` is put into M_n. As a result, method x is found as the direct caller of `Foo#add`.

Step 4: Prune the Call Graph. The loop (lines 11–14) removes illegal CG edges. For updated methods or affected callers, line 13 retrieves their outgoing edges and line 14 removes these stale edges from the new call graph. We only need to remove each stale edge one time. An updated method can be unreachable in the old program, and the corresponding $\mathbb{G}_\Delta$ will be empty. In fig. 1, the `BlackListService#add` is a removed method, and its outgoing edges will be removed after this step.

Step 5: Analyze New Version of Methods. For updated methods, affected callers, and newly reachable methods in the new call graph, lines 16–18 open them to expand the new class hierarchy and the new call graph. Line 17 reuses the `createCH` function to update $\mathbb{H}_n$ and line 18 reuses the `createCG` function to update $\mathbb{G}_n$.



(a) Class hierarchy (b) Call graph

Fig. 5: Program updates to the class hierarchy and the call graph. Red dashed lines denote removed edges and nodes, and green solid lines denote added ones. Gray shadowed squares denote code that is newly reachable in the new program.

Figure 5 presents the updates to the class hierarchy and the call graph in steps 1–4. In the class hierarchy, one node and two edges are removed by step 1[3]. In the first iteration of table II, the following updates happen in order:

1) when visiting ① method, the edge from `VipVerifier` to `Object` is added; the CG edge from `VipService#add` to `VipVerifier#isVip` is added;
2) when visiting ③ method, the inheritance edge from `BlockedService` to `UserService` is added;
3) when visiting ④ method, the new inheritance edge from `FooService` to `Foo` is added;
4) when visiting ⑤ method, the two CG edges are removed, and one new edge to `BlockedService#add` is added.

After the first iteration, the class hierarchy reaches its fixed point. In the second iteration, the method `x` (⑦ in table II) is visited and the edge from it to `FooService#add` is added.

Step 6: Cleanup Unreachable Nodes. The program update can make methods in the old call graph unreachable in the new one. The `cleanUp` in line 20 walks on the new call graph and removes methods that are non-entry methods but have no incoming edges. Their outgoing edges are removed, and the callees are checked to find other unreachable nodes.

The program update can make a reachable type in the old class hierarchy unreachable in the new one. However, these types cannot be removed from the new class hierarchy. One type can be used in multiple places, and analyzing only program updates cannot decide the reachability of the type. We can only know whether one type was used by the old program and can not know whether it is unreachable after the update unless it is a deleted type. Keeping an unreachable type in the new class hierarchy does not affect the soundness of IncCHA because it only leads to some redundant edges and nodes.

B. Tackle the Challenges

The explicit and implicit update effects to call graph are all captured by IncCHA. Explicit effects are introduced by directly updating method invocations. The `processUpdate` function obtains all updated methods, therefore captures all explicit effects. Implicit effects are caused by different virtual call resolutions. Step 3 of alg. 2 captures all implicit effects by locating the unchanged caller methods affected by the update and considering newly reachable methods.

Besides capturing all effects, the carefully designed update sequence in IncCHA preserves the validity and soundness of the class hierarchy and the call graph. Before generating new CG edges in step 5, all invalid inheritance edges were already removed in step 2, so that all constructed CG edges are valid. Before generating new inheritance edges in step 5, all invalid invocation edges were already removed in step 4, so that all constructed inheritance edges are valid. The iterations of IncCHA will finally construct the sound class hierarchy and then the sound call graph, just like the full CHA algorithm.

[3]Here, we assume the superfluous deletions of valid nodes and edges are skipped in optimization.

V. Evaluation

A. Implementation

1) FullCHA: We implemented the full CHA algorithm on top of GraalVM [14]. A large-scale program usually contains two parts: application code and dependent libraries. We use all methods in application code as entry methods to the full CHA algorithm, which means all application code and only reachable code in libraries are analyzed.

2) IncCHA: We implemented two versions of the incremental algorithms, IncCHA^s and IncCHA^b, to handle the case without or with the old bytecode.

The old bytecode may be unavailable for the following reasons. Compiling two versions of program can take a very long time, which is unbearable in CI/CD, or it is unable to compile the old program which is incompatible with the current CI/CD pipeline. IncCHA^s parses the source code patch to over-approximate updated methods. Updated classes can be obtained from the source code patch. Their old methods and super types can be retrieved from the old call graph and class hierarchy, and their new methods and super types can be found in the new bytecode, which is available. The method body can not be compared, therefore all signature-paired methods in changed classes are treated as changed methods.

A program update can also upgrade dependent libraries. Without the old bytecode, we compare the library names (*i.e.*, Jar file names) to determine removed and added libraries, which are available in configuration (*e.g.*, `pom.xml` in Maven project). When saving the class hierarchy in FullCHA, we record an extra attribute for each class: the library name containing the class. Classes from removed (added) libraries are treated as removed (added) classes, and their methods are removed (added) methods.

The `createCH` is also different in IncCHA^s. Without the old bytecode, `createCH` can not get all directly used types for an old method. Only types in method signatures are available. As a result, some removed types may be ignored. If these types are from application code, they will be captured by parsing the source code patch. Otherwise, these types must exist in removed libraries, which are already considered removed types to analyze.

TABLE III: Evaluation results on industrial (1–20) and open-source (21–30) programs: byte-buddy, commons-bcel, commons-io, commons-lang, easyexcel, fastjson, flyway, gson, jedis, and opennlp

ID	Size		PR	Com	LOC$^{\Delta}$(K)		Met$^{\Delta}$		FullCHA		IncCHA^{s}		IncCHA^{b}	
	Jar(MB)	LOC(K)			Max	Avg	Max	Avg	M(GB)	T(S)	M(GB)	T(S)	M(GB)	T(S)
1	176	684.9	43	1110	8.2	0.9	1023	130	11.7	693.8	5.4(46.3%)	54.0(19.5)	5.0(43.2%)	28.4(28.9)
2	313	671.2	44	4161	19.4	4.0	4591	673	16.2	1254.6	10.0(61.6%)	127.0(16.1)	9.1(56.1%)	74.3(22.5)
3	244	433.2	91	5067	18.9	2.7	1502	232	10.0	582.3	4.8(48.6%)	46.0(20.5)	4.3(43.2%)	21.9(29.9)
4	118	277.5	43	1513	14.7	1.3	995	92	5.7	249.4	3.1(55.0%)	22.6(14.4)	2.9(50.8%)	13.7(19.8)
5	226	487.9	51	3123	5.1	1.2	433	116	6.2	282.5	3.5(56.2%)	32.4(11.7)	3.0(49.0%)	14.5(20.6)
6	178	172.2	59	1511	9.3	0.8	9129	587	6.8	322.7	3.6(53.3%)	29.6(16.0)	3.4(51.0%)	21.6(19.2)
7	120	119.8	54	948	5.0	0.5	728	95	4.6	220.0	2.6(57.0%)	16.0(15.3)	2.4(52.2%)	10.9(20.7)
8	435	339.0	50	2142	0.2	0.1	20	10	18.7	1320.6	10.0(53.4%)	68.0(19.9)	9.5(50.9%)	47.2(28.2)
9	201	200.7	79	2456	30.4	1.8	379	113	12.2	585.9	6.2(50.7%)	59.3(12.9)	5.5(45.4%)	30.6(20.5)
10	69	284.9	50	2942	7.0	1.5	725	131	5.6	302.1	3.1(55.5%)	22.2(17.5)	2.9(51.0%)	14.1(23.4)
11	63	214.2	52	1559	2.3	0.3	142	25	3.1	134.9	1.9(63.0%)	10.4(16.2)	1.7(56.7%)	7.4(20.4)
12	374	64.6	57	6672	52.3	6.0	2638	436	12.7	826.8	6.0(47.3%)	68.8(17.6)	5.6(44.1%)	37.2(27.1)
13	115	30.2	70	1476	5.8	0.9	962	137	9.0	414.4	4.4(49.4%)	30.4(17.6)	4.2(46.8%)	22.2(20.8)
14	150	221.3	50	1496	16.3	2.4	1282	210	3.7	204.0	2.3(63.2%)	14.7(18.9)	2.1(56.3%)	9.8(25.9)
15	224	422.5	47	1146	8.1	1.4	913	194	11.8	722.3	5.7(48.8%)	55.8(20.5)	5.4(46.4%)	37.6(26.4)
16	195	810.7	127	3172	22.6	2.7	1370	233	16.6	995.6	7.8(47.9%)	67.8(21.6)	7.3(45.1%)	43.9(26.1)
17	130	243.3	9	1579	31.6	2.1	3605	231	6.5	365.9	3.5(53.9%)	28.5(19.4)	3.2(49.4%)	15.5(26.0)
18	164	155.5	68	1828	9.7	0.8	880	84	8.5	498.0	4.2(49.9%)	27.5(21.9)	4.0(47.6%)	20.2(27.4)
19	213	434.1	29	6080	29.0	0.6	18K	246	11.9	712.1	5.3(45.3%)	41.2(22.7)	5.1(43.5%)	28.6(28.6)
20	186	317.8	45	740	10.2	1.0	905	105	5.1	225.7	2.9(57.8%)	21.3(11.7)	2.6(50.8%)	12.0(19.9)
21	42	170.7	248	6655	4.5	0.7	695	108	4.8	188.7	3.3(69.7%)	43.0(7.3)	2.9(61.0%)	24.3(10.3)
22	3	60.7	25	1964	11.4	2.6	7293	787	1.0	31.3	0.9(94.1%)	4.3(9.0)	0.6(64.0%)	2.3(21.2)
23	2	26.9	58	3955	17.	1.4	582	61	0.7	12.3	0.6(97.1%)	2.5(5.9)	0.5(81.4%)	1.5(13.1)
24	2	68.2	85	7182	40.	3.9	862	106	0.7	13.5	0.7(97.5%)	3.5(4.8)	0.6(80.4%)	1.9(11.2)
25	32	19.4	39	879	17.	1.5	16K	933	1.3	40.9	1.3(94.8%)	8.7(6.1)	1.0(79.0%)	5.0(11.2)
26	55	51.9	132	7203	1.6	0.5	505	68	0.9	23.0	0.9(103.6%)	5.8(4.3)	0.7(79.6%)	3.4(7.6)
27	135	50.1	147	3406	4.3	0.6	31.	198	2.2	86.0	1.6(74.6%)	11.9(9.6)	1.4(64.6%)	6.2(16.7)
28	1	15.1	45	2973	29.7	2.1	89	121	0.6	11.7	0.7(111.6%)	3.4(4.0)	0.6(95.0%)	2.0(7.9)
29	6	42.2	81	2826	64.5	4.3	3K	1545	0.8	17.9	0.9(110.4%)	7.0(3.3)	0.7(86.1%)	4.4(5.1)
30	10	90.1	40	2796	11.9	2.7	10	655	1.4	45.2	1.2(85.8%)	9.5(7.3)	1.0(71.2%)	5.1(14.8)
Avg	**139**	**264.7**	**78**	**3022**	**20.4**	**2.1**	**34.0**	**289**	**6.7**	**379.5**	**3.6(66.9%)**	**31.4(13.8)**	**3.3(58.1%)**	**18.9(20.0)**

The **ID** column presents the program id, 1–20 are from the industrial world and their names are hidden intentionally; 21–30 are open-source programs. The **Size** column presents the average size of bytecode (**Jar**) and lines of code (**LOC**). The **PR** and **Com** columns present the number of releases and commits, respectively. The **LOC$^{\Delta}$** and **Met$^{\Delta}$** columns present the max (**Max**) and average (**Avg**) number of changed code lines and Java methods among the updates we used in the evaluation. The last six columns show the memory (**M**) and time (**T**) different CHA algorithms took to execute. The "m(n%)" in the memory column means that the incremental CHA needs "m" GB of memory, which is "n%" of the memory that the full CHA needs (the lower, the better). The "m(n)" in the time column means that the incremental CHA needs "m" seconds, which is "n" times faster than the full CHA (the higher, the better).

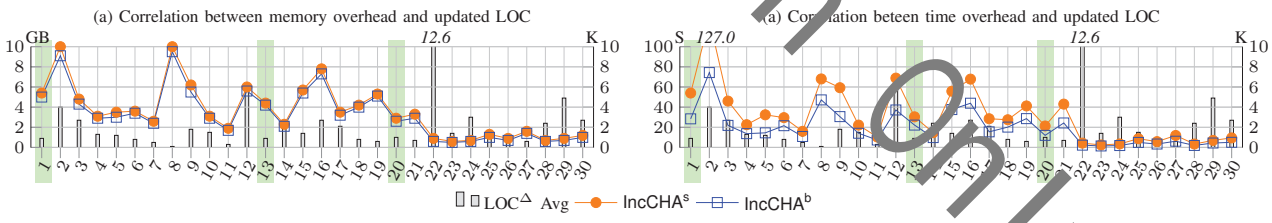


Fig. 6: Correlation between the overhead of incremental construction (the left y-axis) and the updated LOC (the right y-axis). The x-axis is the program id in table III. Green marks programs with similar updated LOC (0.9 ~ 1.1 K) but different overheads.

B. Experimental Setup

We evaluate the effectiveness and efficiency of incremental CHA algorithms by applying FullCHA, IncCHA^{s}, and IncCHA^{b} on target programs to build the class hierarchy and call graph, and comparing their performance.

Table III presents the information of all 30 subjects, including 20 industrial and 10 open-source programs. From the 108 industrial programs in fig. 2, we randomly sampled 20, collected their updates within the last year, and categorized updates according to the distribution in fig. 2. Finally, we randomly sampled one-third of the updates in each distribution and got 433 program updates (out of 1,433). Following the same procedure, we randomly sampled 10 open-source programs (out of 20) and 302 updates (out of 900).

For each update, we first execute FullCHA to build the class hierarchy and call graph for both versions of the update. The results of the old version are used as inputs to incremental CHA, and the results of the new version are used as the comparison criteria of incremental CHA. To evaluate the effectiveness, we compare the CG constructed by incremental

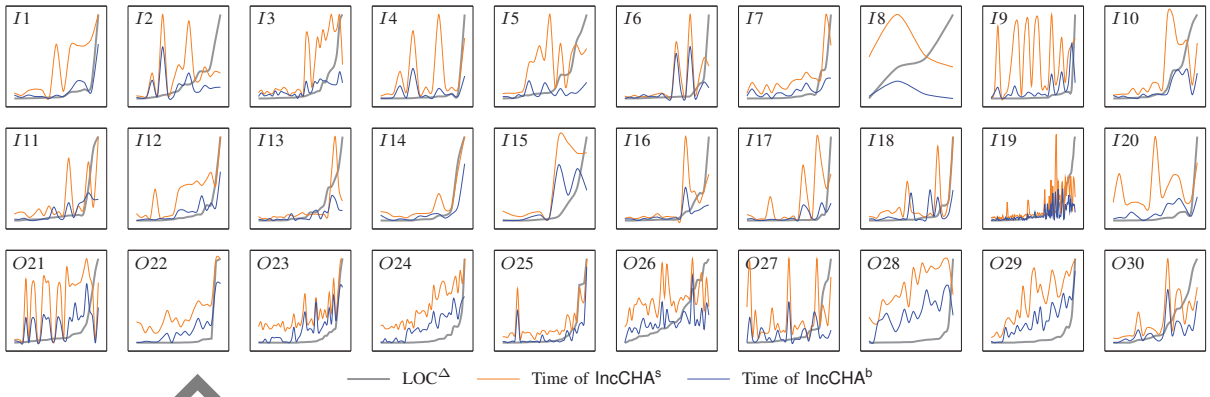


Fig. 7. The number of updated LOC and the incremental construction time of each program update.

CHA with the ones constructed by FullCHA. To evaluate the efficiency, we record the memory, time, and storage space each CHA tool requires for the new program. We run each tool on a program update 5 times and compute the average data.

We set the 1-hour time limit and 25 GB memory limit for each execution. The three CHA tools successfully constructed CGs for all subjects. All experiments were conducted on a PC running CentOS with an Intel E5 CPU and 256 GB RAM.

C. Overall Results

1) FullCHA: The FullCHA column presents the average memory and time it took to execute on the new version for all updates in a program.

The memory and time overhead of FullCHA are related to the program size. Among all 30 programs[4], industrial program *I*8 requires the most memory (18.7 GB) and the most time (1320.6 seconds). Its compiled Jar is the largest among all programs; hence, its CG has the most edges, 16.8 million on average. Open-source program *O*28 (gson [15]) requires the least memory (0.6 GB) and the minimum time (11.7 seconds). Its LOC and the compiled Jar are the smallest among all programs, and its CG only has 119.8 K edges on average.

Industrial programs are more complicated than open-source programs. FullCHA takes more resources to execute on industrial programs. On average, FullCHA takes 9.3 GB and 545.7 seconds to run on industrial programs and only takes 1.4 GB and 47.1 seconds to run on open-source programs.

2) IncCHA^s and IncCHA^b: The IncCHA^s and IncCHA^b columns present the average memory and time they took to execute, along with the average efficiency improvement. IncCHA^b analyzes less updated methods. Therefore it requires less memory and time on most updates than IncCHA^s. On average, IncCHA^b takes 3.3 GB and 18.9 seconds, whereas IncCHA^s takes 3.6 GB and 31.4 seconds. In terms of efficiency improvement, IncCHA^b only needs 58.1% of the memory FullCHA needs and speeds up the construction by 20.0 times. Although IncCHA^s performs slightly worse than IncCHA^b, it still only needs 66.9% of the memory FullCHA needs and speeds up the construction by 13.8 times.

[4]From now on, we use *I*x denote industrial program x, use *O*y denote open-source program y.

Incremental CHA only saves the constructed partial graph (*i.e.*, the patch) and reduces the storage space. The data is not shown in this paper due to space limits. On average, FullCHA needs 140.5 MB to save the full class hierarchy and call graph, whereas IncCHA^s needs 18.4 MB (13.1% of FullCHA) and IncCHA^b needs 14.6 MB (10.4%) to save the graph patch.

D. Detailed Analysis

1) Correlation Between Overhead and Updated LOC: There is an unobvious correlation between the overhead of incremental CHA and the updated LOC among all programs, just as shown in fig. 6. The 3 green shaded programs have similar updated LOC on average. However, their average overheads are not close, especially the memory overhead. *I*8 has the smallest updated LOC, but its memory overhead of IncCHA^s is the highest. The reason is that each program has different basic overhead, such as loading the old graphs. Updating the same LOC among different programs does not introduce similar overhead during incremental construction. However, more updated LOC generally requires more overhead.

The *O*22 (commons-bcel [16]) has the most updated LOC. Several updates renamed its package names for almost all classes so that the updated LOC almost doubled the size of the original program. IncCHA^b still speeds up the construction by 21.2 times on average. That is because part of CG constructed by FullCHA is for library code and can be reused directly during incremental construction. So that IncCHA^s and IncCHA^b only need to rebuild the partial CG for application code.

Figure 7 presents the time of incremental construction with the updated LOC of each update in different programs. The general trend within each program is that the more updated LOC, the longer the construction time. Some updates have less updated LOC but consume longer construction time, such as *I*2 and *O*21. The updated LOC only considers the diff patch of application code and ignores the updates to dependent libraries. Upgrading libraries could introduce more construction time, especially for IncCHA^s, which treats all classes in updated libraries as updated classes.

IncCHA^b runs faster than IncCHA^s on most updates, except several updates in *I*11, *I*13 and *O*27. For these updates, IncCHA^b generates more extra CG nodes and edges than

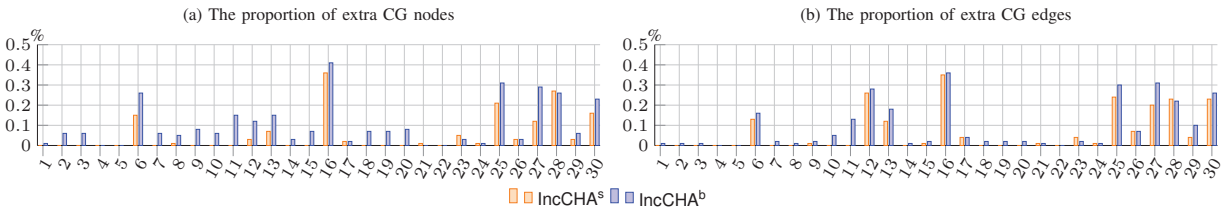


Fig. 8: Extra CG nodes and edges generated by incremental CHA. The x-axis is the program id in table III.

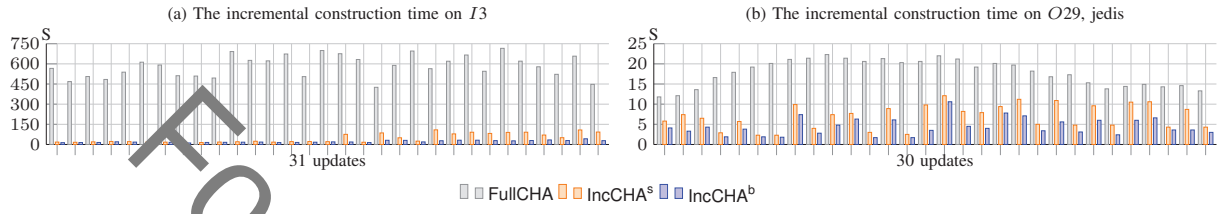


Fig. 9: Incremental CHA speeds up the most (a) and least (b) times. The x-axis is the program updates.

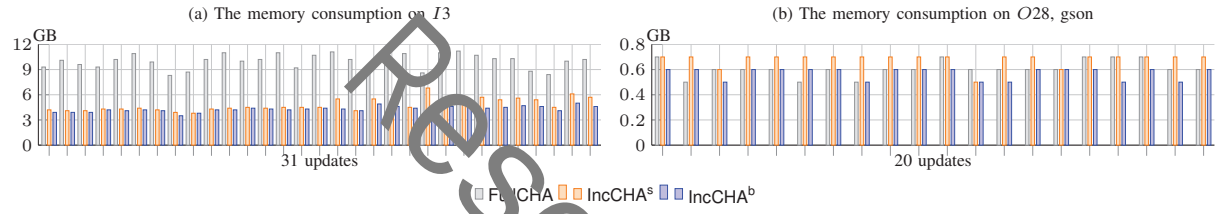


Fig. 10: Incremental CHA reduces memory consumption the most (a) and least (b). The x-axis is the program updates.

IncCHA^s (explained in section V-D2), so that IncCHA^b takes more time (less than 5 seconds) to execute.

2) Extra Nodes and Edges: Figure 8 shows the proportion of extra CG nodes and edges generated by IncCHA^s and IncCHA^b. On average, IncCHA^s generated 34.2 more nodes and2,439.6 more edges; IncCHA^b generated 88.1 more nodes and 3,131.7 more edges. For both tools, the proportions are less than 0.4%, except IncCHA^b generated 0.41% extra nodes for $I16$. Such a proportion is tiny and acceptable, considering the performance improvement incremental CHA can make.

In general, IncCHA^b generates more extra nodes and edges. IncCHA^s treats all classes from upgraded libraries as removed classes and deletes them directly from the class hierarchy. In contrast, IncCHA^b only deletes real removed classes from the class hierarchy (because of line 20). Assume a library class exists in both versions of the program and is only used in the old program, IncCHA^s can delete this class if the library containing the class is upgraded, but IncCHA^b can not. So that IncCHA^s introduces fewer unreachable types into the class hierarchy and generates fewer unreachable CG nodes and edges.

These extra nodes and edges are unreachable but still *valid* in the new program. Hence IncCHA^s and IncCHA^b still ensure the validity property section II-B2. Such extra nodes and edges can be cleaned up by constructing a full CG periodically for the latest version of the program.

3) Time Efficiency: The improvement of time efficiency on industrial programs is better than that of open-source programs, although the average construction time on the latter is less than that of the former. This is because FullCHA took much more time to execute on industrial programs.

Figure 9 shows the construction time on two programs. Among all targets, IncCHA^b speeds up the most (29.9) on $I3$, and IncCHA^s speeds up the worst (3.3) on $O29$. The incremental construction time of $I3$ is higher than the full construction time of $O29$, but the full construction time of $I3$ is even higher, so the improvement on I3 is better.

4) Memory Efficiency: The improvement in memory efficiency is not as good as the improvement in time. There is a basic memory consumption for initializing an instance of VM, loading the old class hierarchy and the old call graph, *etc*. Figure 10 shows the improvements in memory efficiency on two programs. On average, IncCHA^b needs the least proportion (43.2%) of memory on $I3$, and IncCHA^b needs the most proportion (111.6%) of memory on $O28$.

On open-source programs, the memory overhead of incremental CHA is close to or even higher than FullCHA. Because the incremental CHA tools have to create caches for indexing and removing in the class hierarchy and the call graph, such caches are unnecessary for FullCHA. The size of these caches grows along with the updated LOC. Therefore the memory-saving caches are more pronounced for open-source programs.

VI. Threats To Validity

A major threat concerns the soundness of our incremental CG construction algorithm. IncCHA is a reform of the full CHA algorithm, and they share the underlying ability of parsing code and constructing edges. The final new CG generated by IncCHA consists of two parts. The reused part is inherited from the old CG and soundly constructed by CHA for the code irrelevant to the update. The new part is soundly constructed by analyzing the code relevant to the update, *i.e*., updated or affected. The two parts of code make a complete new program. Hence the two parts of sound CG make a sound new CG. The extra nodes and edges (section V-D2) do not affect the soundness of the new CG.

Another threat concerns the evaluation results, because we mainly use programs from Ant Group in evaluation. We argue that (1) because of the confidentiality in industrial world, it is difficult to obtain code from other companies, (2) the evaluation on the 10 open-source programs also shows that IncCHA significantly improves the construction efficiency, and (3) the design and implementation of the target programs follow the general principles of software systems and are representative. Therefore we can conclude that IncCHA indeed constructs CG more efficiently for similar complicated programs.

VII. Related Work

Call graph construction has been studied extensively in the static analysis community. We first review the research focusing on different targets of CG construction, then briefly discuss other incremental analyses.

Resolve Virtual Calls. Reachability analysis (RA) [17] is a simple CG construction algorithm that takes into account only the name (or signature) of a method to resolve virtual calls. Dean *etc*. [4] present CHA which extends RA to use type inheritance (class hierarchy) to determine virtual targets. RTA [12] is a refinement of CHA that narrows down the possible types of a receiver object to subtypes instantiated within the target program, rather than all subtypes in the class hierarchy.

Tip and Palsberg [18] use a unified framework for propagation-based CG construction algorithms. They present four new algorithms (CTA, MTA, FTA, and XTA) that use a distinct set of types to different scopes (*i.e*., classes, methods, fields, or their combinations) to filter possible types further. Sundaresan *etc*. [13] propose VTA and DTA, which find the types that reach each variable. They can be considered refined versions of RTA. The k-CFA [19] adopts points-to analysis with various levels of call-site sensitivity to construct CG, and many modern points-to analyses [18], [19] construct CG on-the-fly.

These algorithms are designed in the exhaustive style, whereas IncCHA is an exploratory study towards incremental CG construction. The incrementalization of these algorithms can reform IncCHA. For incremental RTA, we need to update the set of instantiated types while pruning and updating the class hierarchy. For algorithms that compute propagated types, the stale propagations should be first pruned, and new propagations should be computed after each iteration of updating CG. For algorithms based on points-to analysis, the incrementalization should combine the insight of IncCHA and the incremental points-to analysis [10], [20].

Resolve Other Language Features. Many other language features also affect the construction of CG, especially the dynamic ones [21], [22]. Agesen [23], [24] and Petrashko *etc*. [25] present CG construction algorithms concerning parametric polymorphism. Santos *etc*. [26] provide a serialization-aware CG construction algorithm. There are several CG construction algorithms for dynamic languages, *e.g*., Python [27], [28] and JavaScript [29], [30]. Other feature handling techniques, such as reflection [31], invokedynamic [32], and dynamic proxy [33], all contribute to the precision of CG construction. They are generally orthogonal to this paper. Combining them with incremental construction can also improve the precision of CG.

Improve Efficiency of CG Construction. Demand-driven analysis [34]–[36] only constructs CG when required, and distributed analysis [37] divides the computation to improve the construction speed. Modular analysis [30], [38], [39] constructs CG in a compositional way. They can cooperate with IncCHA to make CG construction even faster.

Souter and Pollock [40] present an incremental version of CPA [24], which transforms echo source editing into adding and deleting a call-site. Lin *etc*. [9] incrementalizes CHA for AspectJ software, which uses atomic change representation to capture the semantic differences between two program versions. Wang *etc*. [41] investigate the influences of edge instability on change propagation and connectivity in CG and [42] present an incremental CG construction algorithm for CG visualization. These researches try to improve efficiency in different domains, whereas IncCHA can be utilized generally in different scenarios.

Other Incremental Analyses. There exist incremental analyses for improving the efficiency in points-to analysis (PTA). Most of them assume a prebuilt call graph before performing the incremental analysis [43], [44], which are orthogonal to this paper and can cooperate with IncCHA. Recent work [10], [20] can incrementally update the call graph along with incremental points-to analysis. However, they require much more resources to execute (*e.g*., 100 GB memory consumption in [10]), which is impractical in industrial scenarios.

VIII. Conclusion

This paper revisits the need for efficient CG construction for CI/CD pipelines in the industrial world, recognizes the challenges of incremental CG construction, then proposes the IncCHA algorithm to patch the previous CG to construct a new one. The experimental evaluation shows that IncCHA can efficiently construct valid and sound CG for industrial and open-source programs. Compared with the full construction algorithm, IncCHA can speed up the construction by 20.0 times, reduce the memory consumption to 58.1%, and only need 10.4% storage space.

References

[1] X. Zhang, R. Mangal, M. Naik, and H. Yang, "Hybrid top-down and bottom-up interprocedural analysis," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2014, pp. 249–258. https://doi.org/10.1145/2594291.2594328

[2] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*. Pearson Education, 2004.

[3] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[4] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP)*. Springer, 1995, pp. 77–101. https://doi.org/10.1007/3-540-49538-X_5

[5] P. W. O'Hearn, "Continuous reasoning: Scaling the impact of formal methods," in *Proceedings of the 33rd annual ACM/IEEE symposium on logic in computer science (LICS)*. ACM, 2018, pp. 13–25. https://doi.org/10.1145/3209108.3209109

[6] N. Chong, B. Cook, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Code-level model checking in the software development workflow," in *Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2020, pp. 11–20. https://doi.org/10.1145/3377813.3381347

[7] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 598–608. https://doi.org/10.1109/ICSE.2015.76

[8] U. Ismail, "Incremental call graph construction for the eclipse ide," Tech. Rep., 2009.

[9] Y. Lin, S. Zhang, and J. Zhao, "Incremental call graph reanalysis for aspectj software," in *Proceedings of the 2009 IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2009, pp. 306–315. https://doi.org/10.1109/ICSM.2009.5306311

[10] B. Liu, J. Huang, and L. Rauchwerger, "Rethinking incremental and parallel pointer analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 41, no. 1, pp. 6:1–6:31, 2019. https://doi.org/10.1145/3293606

[11] Maven. Accessed: 2022-09-22. https://maven.apache.org

[12] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 1996, pp. 324–341. https://doi.org/10.1145/236337.236371

[13] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2000, pp. 264–280. https://doi.org/10.1145/353171.353189

[14] GraalVM. Accessed: 2022-09-22. https://graalvm.org

[15] Gson. Accessed: 2022-09-22. https://github.com/google/gson

[16] Apache Commons BCEL. Accessed: 2022-09-22. https://commons.apache.org/proper/commons-bcel

[17] A. Srivastava, "Unreachable procedures in object-oriented programming," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 4, pp. 355–364, 1992. https://doi.org/10.1145/161494.161517

[18] F. Tip and J. Palsberg, "Scalable propagation-based call graph construction algorithms," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2000, pp. 281–293. https://doi.org/10.1145/353171.353190

[19] O. G. Shivers, "Control-flow analysis of higher-order languages of taming lambda," PhD Thesis, Carnegie Mellon University, 1991.

[20] B. Liu and J. Huang, "SHARP: fast incremental context-sensitive pointer analysis for java," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 6, no. OOPSLA1, pp. 88:1–88:28, 2022. https://doi.org/10.1145/3527332

[21] M. Reif, F. Kübler, M. Eichberg, D. Helm, and M. Mezini, "Judge: Identifying, understanding, and evaluating sources of unsoundness in call graphs," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2019, pp. 251–261. http://doi.org/10.1145/3293882.3330555

[22] L. Sui, J. Dietrich, A. Tahir, and G. Fourtounis, "On the recall of static call graph construction in practice," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. ACM, 2020, pp. 1049–1060. https://doi.org/10.1145/3377811.3380441

[23] O. Agesen, "Constraint-based type inference and parametric polymorphism," in *Proceedings of the First International Static Analysis Symposium (SAS)*. Springer, 1994, pp. 78–100. https://doi.org/10.1007/3-540-58485-4_34

[24] O. Agesen, "The cartesian product algorithm: Simple and precise type inference of parametric polymorphism," in *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP)*. Springer, 1995, pp. 2–26. https://doi.org/10.1007/3-540-49538-X_2

[25] D. Petrashko, V. Ureche, O. Lhoták, and M. Odersky, "Call graphs for languages with parametric polymorphism," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2016, pp. 394–409. http://doi.org/10.1145/2983990.2983991

[26] J. C. S. Santos, R. A. Jones, C. Ashiogwu, and M. Mirakhorli, "Serialization-aware call graph construction," in *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP)*. ACM, 2021, pp. 37–42. https://doi.org/10.1145/3460946.3464319

[27] G. Gharibi, R. Tripathi, and Y. Lee, "Code2graph: automatic generation of static call graphs for python source code," in *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE)*. ACM, 2018, pp. 880–883. http://doi.org/10.1145/3238147.3240484

[28] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "Pycg: Practical call graph generation in python," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1646–1657. https://doi.org/10.1109/ICSE43902.2021.00146

[29] D. Seifert, M. Wan, J. Hsu, and B. Yeh, "An asynchronous call graph for javascript," in *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2022, pp. 29–30. https://doi.org/10.1109/ICSE-SEIP55303.2022.9794057

[30] B. B. Nielsen, M. T. Torp, and A. Møller, "Modular call graph construction for security scanning of node.js applications," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2021, pp. 29–41. https://doi.org/10.1145/3460319.3464836

[31] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 2, pp. 7:1–7:50, 2019. https://doi.org/10.1145/3295739

[32] G. Fourtounis and S. Yan..., "Deep static modeling of invokedynamic," in *Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 15:1–15:28. https://doi.org/10.4230/LIPIcs.ECOOP.2019.15

[33] G. Fourtounis, G. Kastrinis, and Y. Smaragdakis, "Static analysis of java dynamic proxies," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2018, pp. 209–220. https://doi.org/10.1145/3213846.3213864

[34] G. Agrawal, "Simultaneous demand-driven data-flow and call graph analysis," in *Proceedings of the 1999 IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 1999, pp. 453–462. https://doi.org/10.1109/ICSM.1999.792643

[35] G. Agrawal, "Demand-driven construction of call graphs," in *Proceedings of the 9th International Conference on Compiler Construction (CC)*. Springer, 2000, pp. 125–140. https://doi.org/10.1007/3-540-46423-9_9

[36] G. Agrawal, J. Li, and Q. Su, "Evaluating a demand driven technique for call graph construction," in *Proceedings of the 11th International Conference on Compiler Construction (CC)*. Springer, 2002, pp. 29–45. https://doi.org/10.1007/3-540-45937-5_5

[37] D. Garbervetsky, E. Zoppi, and B. Livshits, "Toward full elasticity in distributed static analysis: the case of callgraph analysis," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 442–453. http://doi.org/10.1145/3106237.3106261

[38] D. Helm, F. Kübler, M. Reif, M. Eichberg, and M. Mezini, "Modular collaborative program analysis in OPAL," in *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2020, pp. 184–196. https://doi.org/10.1145/3368089.3409765

[39] P. D. Schubert, B. Hermann, and E. Bodden, "Lossless, persisted summarization of static callgraph, points-to and data-flow analysis," in *Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 2:1–2:31. https://doi.org/10.4230/LIPIcs.ECOOP.2021.2

[40] A. L. Souter and L. L. Pollock, "Incremental call graph reanalysis for object-oriented software maintenance," in *Proceedings of the 2001 IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2001, pp. 682–691. https://doi.org/10.1109/ICSM.2001.972787

[41] L. Wang, H. Li, and X. Wang, "The influences of edge instability on change propagation and connectivity in call graphs," in *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 2016, pp. 197–213. https://doi.org/10.1007/978-3-662-49665-7_12

[42] I. Császár and R. R. Slavescu, "Interactive call graph generation for software projects," in *Proceedings of the 2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing (ICCP)*. IEEE, 2020, pp. 51–58. https://doi.org/10.1109/ICCP51029.2020.9266149

[43] Y. Lu, L. Shang, X. Xie, and J. Xue, "An incremental points-to analysis with cfl-reachability," in *Proceedings of the 22nd International Conference on Compiler Construction (CC)*. Springer, 2013, pp. 61–81. https://doi.org/10.1007/978-3-642-37051-9_4

[44] S. Arzt and E. Bodden, "Reviser: efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 288–298. https://doi.org/10.1145/2568225.2568243