On the Effectiveness of Incremental Fact Extraction & Analysis

by

Davood Anbarnam

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Today's software projects can be huge. They often consist of millions of lines of code, have multiple teams working on them and are constantly evolving. It is no surprise then that developers sometimes seek the help of advanced diagnostic tooling, such as static analysis tools, to aid the development process, with many modern Integrated Development Environments (IDEs) such as Eclipse and Visual Studio providing such functionality out-of-the-box. Fact Extraction is one such static analysis technique that extracts a base model of the underlying software containing properties of the system entities (e.g., variables, functions, files/classes) and their relationships, and stores them in the form of a database of facts (**factbase**). This base model can then be queried and analysed by developers to reveal higher-level design information, such as dataflow between various modules of a software system.

Currently, approaches to building system models scale fairly well to large single systems; factbases can be created using time and resources comparable to that of the compilation process. However, software systems evolve over time, and these analyses need to be redone as the source code changes. While incremental compilation techniques have the potential to greatly reduce the time taken to rebuild the systems themselves, as yet there has been little research into tools that support incremental analysis of changing artifacts to produce revised factbases. This thesis proposes an extraction and analysis framework that is more amenable to creating models from changing artifacts: an Incremental Extraction and Analysis Framework. In particular, we focus on how changes at the file level — i.e., modifications to source files as well as the addition and removal of source files — can impact a previously extracted model and analysis.

We evaluate our proposed framework by performing a case study on a build of the Linux Kernel. First, we compare two approaches to extracting a revised factbase from a new version of the build: one that uses a traditional approach and one that uses our proposed incremental extraction techniques. Then, we compare two approaches to analysing revised factbases: one that uses an incremental approach and one that does not. We found that significant performance improvements can be made in extracting a revised factbase when using an incremental approach, with extraction times being reduced by at least 50%, while re-analysing a revised factbase using an incremental approach grows linearly in terms of the number affected facts in the best case and follows an S-shaped growth in the worst case. We found that the cause for the observed exponential growth could be traced to a subset of facts, rather than being the result of a gradual increase of an analysis' search space.

Acknowledgements

First and foremost, I would like to thank Mike Godfrey for being an exceptional supervisor. I appreciate the freedom you've given me to make mistakes during this process, and the patience you've shown in helping me learn from them.

I would also like to thank my readers Jo Atlee and Chengnian Sun for their valuable feedback, as well as Mei Nagappan.

Thank you to all the wonderful people of SWAG (Aaron, Achyudh, Arman, Aswin, Bushra, Cassiano, Charles, Cosmos, Dan, Gema, Kilby, Laksh, Magnus, Oscar, Reza, Shameer, Shiva, Ten, Yiwen) and the Lightweight Analysis Group (Rob, Rafael, Finn, Sunjay, Jeremy). My time in the lab was infinitely more fun thanks to you. All of you truly are brilliant.

Last, but not least, thank you to my parents. None of this would be possible without you.

Dedication

To my parents, Mitra and Aziz.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Successful software projects often span multiple teams and locations, consist of millions of lines of code and become increasingly complex as they age and evolve [30]. The Linux Kernel, for example, grew from 176,250 lines of code in its 1.0.0 release in 1994 [5] to 15 million lines of code with over 1,300 individual contributors in 2011 [4], and then to 19 million lines of code with around 14,000 contributors in 2015 [3]. These challenges make it increasingly difficult for developers to produce reliable, robust and correct pieces of software, sometimes leading to costly failures. Japan's Hitomi satellite experienced a software failure that caused it to spin out of control five weeks after launch, $USD 286M being lost in the process [46]. In 2015 it was discovered that an integer overflow could potentially cause a Boeing 787's electrical system to completely shut off [6], while in 2019 a technical fault was causing Lime e-scooters in Auckland to suddenly break at high speeds [1].

One technique that can be used to mitigate such failures is static analysis, a family of automated approaches that leverage source code artifacts to help verify properties of the software system without having to execute it. For instance, medical device manufactures are now leveraging static analysis tools to help verify their safety-critical products against errors such as buffer overflow [2], while Google uses its in-house tool Error Prone[1], an extension of the `javac` compiler, to help detect common Java bugs at compile-time [41]. For example, an EqualsWrongThing[2] error is detected when different pairs of class fields or class getters are compared in a class' `equals` implementation. At the most basic level, a study by Prause et al. found that 60% of interviewed developers involved in European

[1] https://github.com/google/error-prone

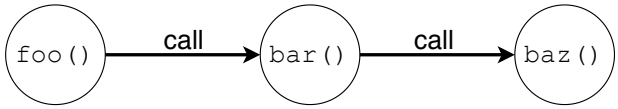
[2] https://errorprone.info/bugpattern/EqualsWrongThing

research projects used their IDE's built-in static analyzers [36]. Our work focuses on a specific type of static analysis: *Fact Extraction and Analysis*.

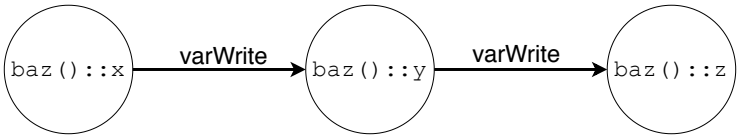
Fact Extraction is the process of extracting a model from a given set of software design artifacts such as source code artifacts. The extracted model contains the properties of software system's entities, such as variables, functions, classes, etc., and their relationships. This model is represented as a database of facts (factbase), each fact being a piece of information that was extracted from a source code artifact. For example, we can extract all direct function calls from the code snippet in Figure 1.1, resulting in the extracted model shown in Figure 1.2a. The model of the system we extract depends to some degree on how we intend to use it. The model in Figure 1.2a, for example, would not be particularly useful if we are interested in dataflow through variables. Accordingly, we can extract all direct variable writes from the code snippet in Figure 1.1, which would result in the extracted model shown in Figure 1.2b.

```
void foo(){
  bar();
}

void bar(){
  baz();
}

void baz(){
  int x = 1;
  int y = x;
  int z = y;
}
```

Figure 1.1: Example Code Snippet to Extract Facts from



(a) Direct Function Calls extracted from Figure 1.1



(b) Direct Variable Writes extracted from Figure 1.1

Figure 1.2: Facts Extracted from Code Snippet in Figure 1.1

Having extracted a factbase from source code artifacts, we can then query/analyse it for higher level facts; that is, we perform a *Fact Analysis* on the extracted facts. For example, we can query the factbase in Figure 1.2a to find all possible indirect calls between the functions `foo()`, `bar()` and `baz()`, or we can analyse the factbase in Figure 1.2b to find all possible indirect variable writes between the variables `baz()::x`, `baz()::y` and `baz()::z`. The results of these queries can then be saved for future use.

Like any other static analysis technique, Fact Extraction and Analysis can be useful to developers only if it is practical for them to use in their day-to-day tasks. A key factor in earning developer buy-in is how easily it can be integrated into a developer's existing workflow [28]. As pointed out by Cadowski et al. [41] and Johnson et al. [28], developers will refrain from using static analysis techniques if they feel it would disrupt their general workflows, even if it does improve the overall quality of the software. With numerous projects taking an Agile approach to software development [8], where developers work incrementally in short sprints, subsequent analyses should be able to fit within this Agile model. It is precisely this area where current Fact Extraction & Analysis approaches can be improved upon.

Our goal in this thesis is to improve the general practicality of the Fact Extraction and Analysis framework by making it more change friendly, which we hope will make it easier to integrate into the developer workflow. In particular, we focus on changes at the source file level — i.e., modifications to source files, addition of new source files and removal of source files — and how they impact a previously extracted model and analysis. We first present an approach that modifies an established Fact Extraction and Analysis toolchain [35]. We then evaluate our approach using our proposed toolchain by performing a case study on the Linux Kernel, using our findings to evaluate the effectiveness of our approach and to map out areas of improvement and future work.

1.1 Thesis Contributions

This thesis presents three major contributions:

- An approach for incrementally updating previously-extracted System Factbases, building upon the Separate Fact Extraction pipeline to account for changes to source files.
- An approach for incrementally updating a set of analysis results using a strategy that is similar to a Rete network. We introduce the notion of *Meta Facts* to allow for such a strategy.
- A case study that evaluates the effectiveness and feasibility of our proposed approaches. The case study was conducted on the Linux kernel source code by modifying an existing extraction and analysis pipeline to incorporate our incremental approaches. We compared the batch and incremental approaches to updating previously extracted factbases, and the batch and incremental approaches to updating

a previous set of analysis results. Overall we found that our Incremental Extraction achieved improvements of 50%-65% in extraction time, while we found our Incremental Analysis to be highly sensitive to subsets of facts in the presence of transitive closure, scaling linearly in the best case and in the worst case following an S-shaped growth as a result.

1.2 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we discuss relevant background knowledge on Fact Extraction and Analysis and its use in the literature. As well, we discuss related work on factbase representations, incremental model extraction and incremental analysis techniques. Chapter 3 describes our approaches to incrementally updating a System Factbase and incrementally updating analysis results, where we introduce the notion of *Meta Facts*. We evaluate these approaches through a case study in Chapter 4, presenting our methodology for the case study, major findings and limitations. Finally in Chapter 5, we summarize the work completed in this thesis and suggest areas of future work.

Chapter 2

Background & Related Work

In this chapter, we give an overview of a typical Fact Extraction and Analysis framework in current use, including previous source code analyses that have leveraged this framework, and existing techniques for updating a previously extracted model and analysis. We start by exploring this framework in Section 2.1 and outline how we represent a factbase in Section 2.2, presenting alternative representations and tools that have been previously used. We discuss how our chosen representation has been used for source code analysis in Section 2.3, while Section 2.4 explores other analyses that have been carried out through Fact Extraction. In Section 2.5, we describe previous approaches to updating an extracted source code model, while we describe how current techniques in Incremental View Maintenance of relational and graph data can be applied to update analysis results in Section 2.6.

2.1 Fact Extraction as Compilation

Fact Extraction & Analysis has two phases: the extraction process and the analysis process. The former can be thought of as a model extraction process, where a parser is used to extract relevant source code information into an internal representation (IR), most commonly a graph, while the latter can be viewed as state-space exploration [11]. In this sense, the extraction process closely follows the well-established compiler pipeline architecture [31]: the Front End parses a given source file, producing a model of the code in an intermediate representation (IR), and then the Back End transforms the IR model into a collection of "facts" about the source code, rather than machine code. We can call this collection of facts a *local factbase*.

Much like the compilation of source code into an executable, there is a final linking step in the extraction of a model from a set of source code. Where traditional compiler linking collects and resolves dependencies between units of generated machine code into a final executable, linking in a Fact Extraction sense collects and resolves dependencies between generated *local factbases* into a final *system factbase* [47]. Let us consider Figures 2.1 and 2.2 as an example. We first extract factbases from the two source files in Figure 2.1 into separate *local factbases*, which for simplicity we represent as a graph. The top factbase has already had its symbols, namely its nodes, resolved, as functions `foo()` and `bar()` and the variable `globFlag` are all declared within the source file. We can call edges between resolved symbols **resolved edges**. On the other hand the bottom factbase contains *virtual* symbols, owing to the `extern` declaration of the functions `foo()` and `lie()`, and these links will have to be resolved during the linking step in Figure 2.2. We can call edges between at least one unresolved symbol an **unresolved edge**. It is during this linking step that we resolve `foo()`, meaning we resolve the edge between `baz()` and `foo()` and keep it in the *system factbase*, but are unable to resolve `lie()`. Such a scenario typically arises when the software system we are extracting from contains functionality from external libraries, e.g., calls to system library functions; we can mark the edge between `baz()` and `lie()` as unresolved and keep it in the *system factbase*, or we can exclude this edge entirely if we consider library functionality to be out of the *system factbase*'s scope, which we do in Figure 2.2.

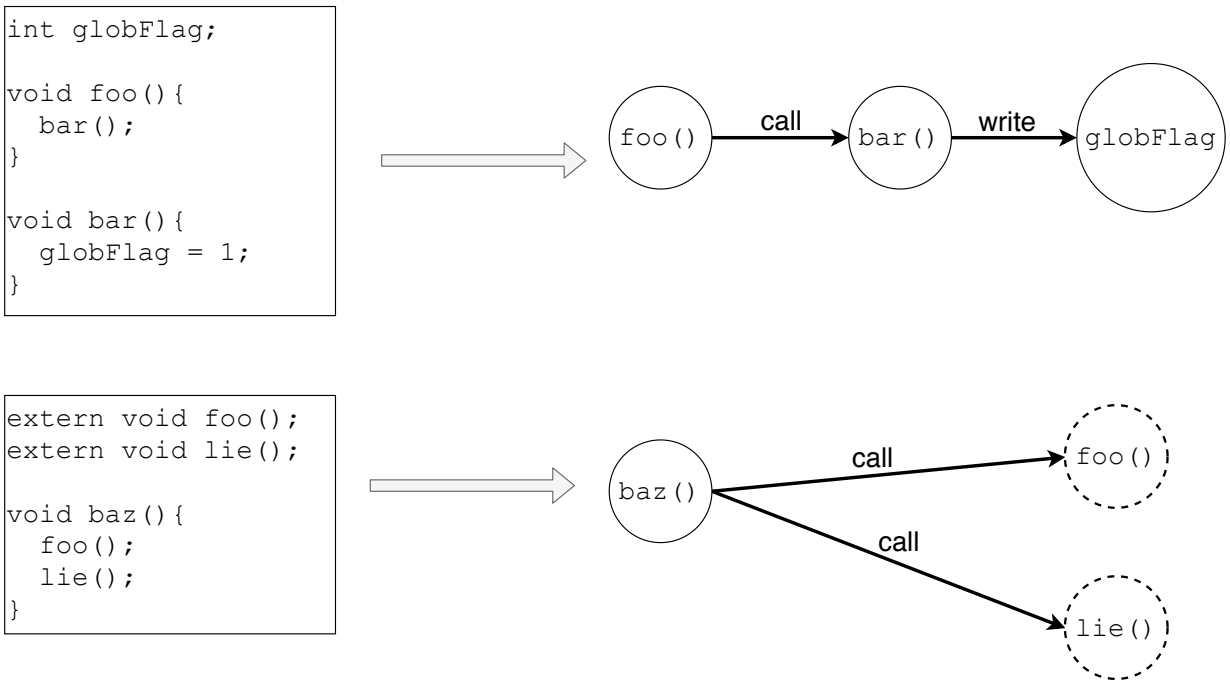


Figure 2.1: Extraction of *Local Factbases*

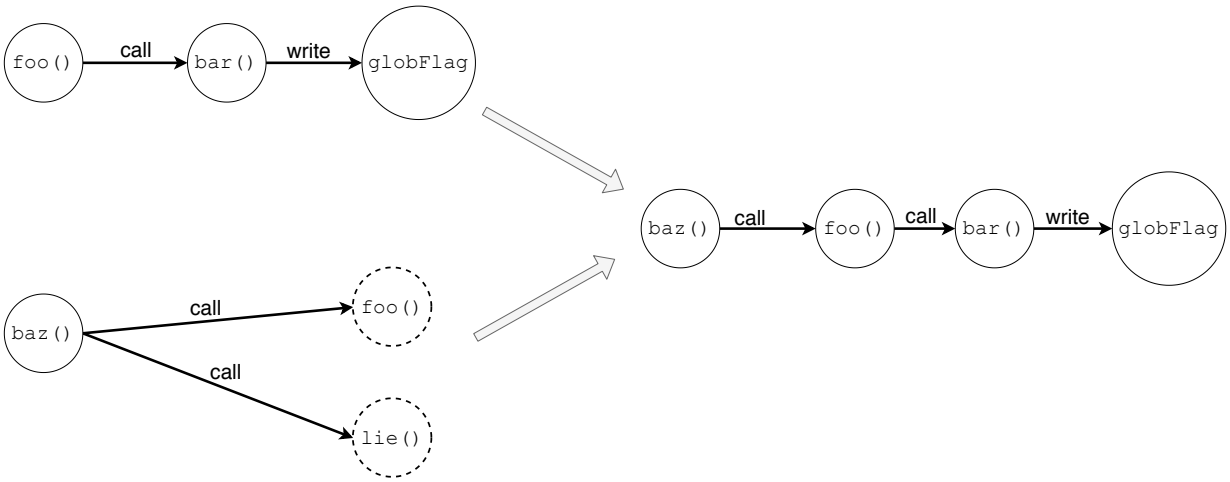


Figure 2.2: Linking Separately Extracted *Local Factbases* into a *System Factbase*

This modular approach makes the overall extraction process more memory and time-efficient on large systems with many source files [15], allowing us to extract *local factbases* in parallel in a manner analogous to separate compilation. We refer to this pipeline, shown Figure 2.3, as the **Separate Extraction Pipeline**; we first extract separate *local factbases* for each given source file during the *Extract* step, and link these factbases into the *system factbase*, *Sys*, during the *Link* step. Once we have our factbase *Sys* extracted, we can then perform analyses on *Sys*, such as dataflow or call flow analysis, in the *Analysis* step, generating a set of derived facts *Analysis(Sys)* to supplement the extracted facts in *Sys*.

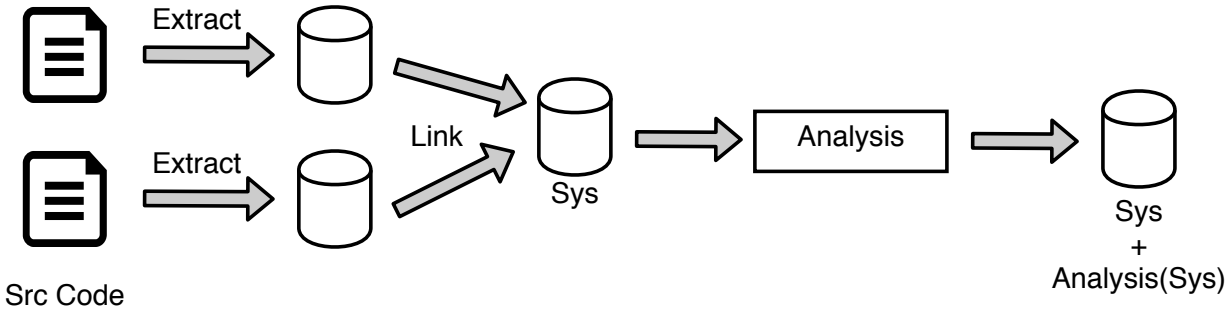


Figure 2.3: Separate Extraction Pipeline

2.2 Factbases as Property Graphs

In this work, we interpret a *system factbase* as a property graph. A property graph is a directed, labelled multigraph where each node and edge maintains a set of key-value pairs called **properties**. Each node can have zero or more **labels**, while each edge can have at most one **label** [7]. We provide an overview of alternative formats that have been used in the context of fact extraction and analysis, and how they represent a *system factbase* using the property graph model, below.

2.2.1 Tuple-Attribute Format

In the Tuple-Attribute (TA) format, coined by Holt [26], facts are represented as a set of ordered, binary relations between entities, each entity instance having a type/label and each entity and relation instance having zero or more attributes as key-value pairs. Figure 2.4 illustrates an example of such a factbase, where entries of the form `$INSTANCE <EntityName>`

`<EntityType>` denote entity declarations and entries of the form `<RELATION> <SOURCE> <DESTINATION>` denote instances of specific relations. Hence, the entry `$INSTANCE globFlag cVariable` denotes that `globFlag` is a variable, while the entry `call foo() bar()` is an instance of the `call` relation that denotes that function `foo()` calls function `bar()`. Alternatively one could use the srcML format [14] to represent a *system factbase* in an XML-like language, which would allow the use of XML tools like XPath to perform any querying/analysis; or the Rigi Standard Format (RSF), which was an inspiration for the TA format.

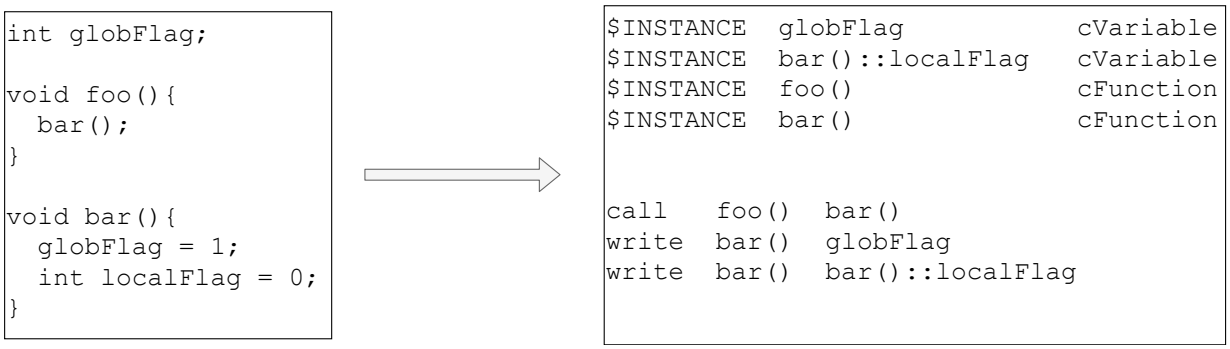


Figure 2.4: An example of a TA factbase

2.2.2 Datalog

We can also represent a *system factbase* as a Datalog program. Datalog is a declarative, relational query language that supports logical inference [22]. A Datalog program is made up of a set of facts, referred to as an Extensional Database (EDB), and a set of inference rules that operate on facts, referred to as an Intentional Database (IDB). The facts in an EDB are defined as n-ary predicate expressions with only constants as their arguments, while each inference rule in an IDB is defined as a Horn Clause of predicate expressions. These rules are repeatedly applied to an EDB by an inference algorithm, generating new facts that are added to the EDB until a fixed point is reach, i.e. no new facts are added to the EDB. In the context of program analysis, we can interpret the predicates in an EDB and IDB as n-ary relations. In this context, Datalog and TA are very similar; a choice between the two is a matter of one's preference for a particular syntax or available tooling. One notable difference is that Datalog supports Incremental View Maintenance techniques [33], while TA does not.

Figure 2.5 shows how we can represent a *system factbase* in Datalog, using the same code snippet as in Figure 2.4. The EDB in this case consists of the same set of facts extracted in

Figure 2.4 in a slightly different syntax. The IDB consists of the `indirectWrite` rule, defined as the composition of the `call` and `write` relation. After applying the inference algorithm, the relation instances `indirectWrite(foo(), globFlag)` and `indirectWrite(foo(), bar()::localFlag)` will have been added to the EDB.

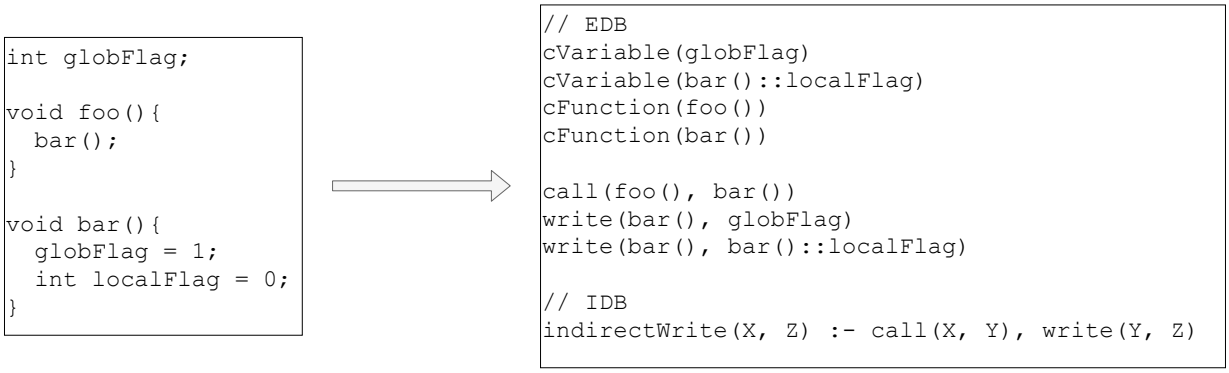


Figure 2.5: An example of a Datalog factbase

2.3 Source Code Models as Property Graphs

The practice of extracting a queryable model from source code artifacts is not new. Recent work in generating such models, however, are increasingly using the property graph model [39], using a tool such as Neo4j. Yamaguchi et al. used the property graph model to discover vulnerabilities in the Linux kernel [48]. To do so, they first modelled the code's Abstract Syntax Tree, Control Flow Graph and Dependency Graph as property graphs, merging the separate representations into one *code property graph*. They then used graph traversal algorithms to find paths that matched specific vulnerability patterns using Apache Gremlin, detecting 18 new vulnerabilities.

Goonetilleke et al. developed a code comprehension tool, Frappe, on top of Neo4j that maintained a versioned dependency graph of a system's codebase, the main goal of the project being to aid developers in the code review process [21]. This versioned graph would allow developers to ask questions that could come up during code review, like which version saw the introduction of a new function or which version saw the addition of a new dependency, using graph traversal queries written in the Cypher query language [18].

Focusing on more niche systems, Prähofer et al. [37] used Static Analysis techniques of Programmable Logic Controller (PLC) programs to extract their structural data and

dependencies into the Neo4j Graph Database, creating an architectural view that users could query using the Neo4j Web Interface to check for violations of design & architectural constraints. Similarly, Thaller's **Gradient** tool used a program's static, structural data as an index for data generated from dynamic analysis, allowing engineers to cross-reference source code entities to their run-time behaviour [45].

2.4 Fact Extraction

The earliest use of Fact Extraction & Analysis can be traced back to Reverse Engineering, specifically the recovery of system architectures. The concept itself was first introduced by Schwanke [42] and later formalized by Murphy et al. as the Software Reflexion Model [34], which detects disparities between the conceptual view of a system, as identified by engineers, and the extracted view of a system's source code. Following this approach, Holt used the Fact Extraction & Analysis framework with the Relational Algebra tool **Grok** to recover the Linux Kernel's architecture [27].

Godfrey et al. developed a prototype system, **Beagle**, that uses the Fact Extraction & Analysis framework to detect merges and splits of functions and files between different versions of a software [20]. Beagle first extracts a factbase from a given version of a software system and when fed with another version of the same system, the tool then performs an origin analysis, annotating the previously extracted factbase with information about the origin of extracted software entities, namely functions. They evalutaed their approach by performing a case study on 11 pairs of successive versions of PostgreSQL, finding that merging and splitting made up 12% of the total structural changes seen in PostgreSQL's evolution between the studied versions.

Muscedere et al. applied Fact Extraction & Analysis to help verify Automotive Software by detecting potential **Feature Interactions** (FIs) [35], using an in-house extractor Rex to build a factbase of software entities and a set of **Grok** analysis scripts to detect potential Feature Interactions, each script detecting a specific type of Feature Interaction. This lighter weight approach to detecting Feature Interactions was evaluated on the Autonomoose project[1], finding 1444 instances of possible Feature Interactions, of which 149 were confirmed by a domain expert to be likely true positives.

Datalog analyses, and more generally Datalog-like analyses, have predominantly focused on *Pointer Analysis*. Benton et al. developed the DIMPLE framework that extracted

[1] https://uwaterloo.ca/centre-automotive-research/watcar-autonomoose/about-autonomoose

a factbase from Java bytecode, which they call a DIMPLE IR, and allowed users to interactively query the factbase through logic programming [9] (in their case Prolog). Lam et al. [29] opted to use Binaray Decision Diagrams to represent a factbase in their bddbddb system to perform Context-Sensitive Analysis, evaluating Datalog queries/analyses as BDD programs, while Bravenboer et al. sought to improve performance times of context-sensitive analysis by heavily optimizing a given Datalog program through their DOOP program [12]. In contrast, Eichberg et al. used a Datalog factbase to enforce architectural-level constraints placed on a software system, applying their approach to the BAT toolkit [16].

2.5 Incremental Model Extraction

Chen's C Information Abstraction System (CIA) was the first foray into incremental construction of a program model [13]. Like Separate Fact Extraction, CIA constructs a system model of a given program from constituent local models, leveraging Unix's *Make* to reconstruct local models and update the system model from source files that have been modified, although the finer details of this approach are not elaborated on. CodeQuest [25] follows a similar approach to updating a system model, using the auto-build feature in Eclipse to first purge all previous facts extracted from compilation units that have been recompiled or removed, and second adds the facts that have been newly extracted from compilation units that have been recompiled (including new compilation units). Stein et al. [44] took a slightly different approach to CodeQuest by using a Version Control System (VCS) to identify affected compilation units rather than a build utility. Our work takes a more finely-grained approach to updating a system model than the ones described; our goal is to reduce the amount of unnecessary work, looking for changes at the fact-level rather than the file-level, particularly in the case of modified source files.

2.6 Incremental Analysis

Incremental Analysis can be viewed as a form of Incremental View Maintenance (IVM) [10]: the View V is a set of analysis results that were derived from an analysis query Q and an underlying set of base facts B in the *system factbase*. IVM is a technique used to update V to V' when B changes to B' using ΔB, rather than computing V' using Q and B' from scratch. Let us consider the *system factbase* in Figure 2.4 as an example. In this factbase, B is the set of facts:

```
call foo() bar()

write bar() globFlag

write bar() bar()::localFlag
```

Suppose we want to carry out the following analysis *indirectWrite*: find all functions that indirectly write to a variable by calling another function. Hence, our analysis query Q would be the composition of the *call* and *write* relations. When we execute Q, V, the results of the *indirectWrite* query will be the set of derived facts:

```
foo() globFlag

foo() bar()::localFlag
```

Now suppose our *system factbase* is updated with `write bar() bar()::localFlag` being removed and `write bar() bar()::x` being added. B' is thus the set of facts:

```
call foo() bar()

write bar() globFlag

write bar() bar()::x
```

and ΔB is the set:

```
- write bar() bar()::localFlag

+ write bar() bar()::x
```

To derive V', which would be:

```
foo() globFlag

foo() bar()::x
```

we can either execute Q on B' or use ΔB to update V incrementally.

Previous work has focused on IVM in the relational data model, with techniques largely centered around using ΔB to construct relational expressions that carry out the update from V to V' [38][23]. These expressions are constructed 'from the ground up'; changes seen to the base relations in Q are used to create an incremental version of the query ΔQ. For example, suppose that Q is $call \circ write$ and that ΔB consists only of changes to the $write$ relation: $write_{added}$ and $write_{removed}$. Using ΔB, we can construct a corresponding ΔQ to carry out the update: $V \cup (call \circ write_{added}) - (call \circ write_{removed})$.

Since we interpret a *system factbase* as a property graph, however, we are more interested in IVM techniques for the property graph model, an area that is significantly less mature than its relational counterpart. The work by Szárnyas looks to address this disparity [19]. Fundamentally his approach is based on evaluating a query using a Rete network [17], which historically has been used in rules-based systems like Datalog. Changes to the graph that are relevant to a query, namely ΔB, are detected at the leaves of the network and propagated towards the root, each propagation step storing intermediate results. An example of this approach for the query $call \circ write$ is shown in Figure 2.6, where our initial set of base facts B is:

```
call foo() bar()

write bar() globFlag

write bar() bar()::localFlag
```

and our initial result set V is:

```
foo() globFlag

foo() bar()::localFlag
```

For simplicity, we have excluded any changes to the *call* relation.

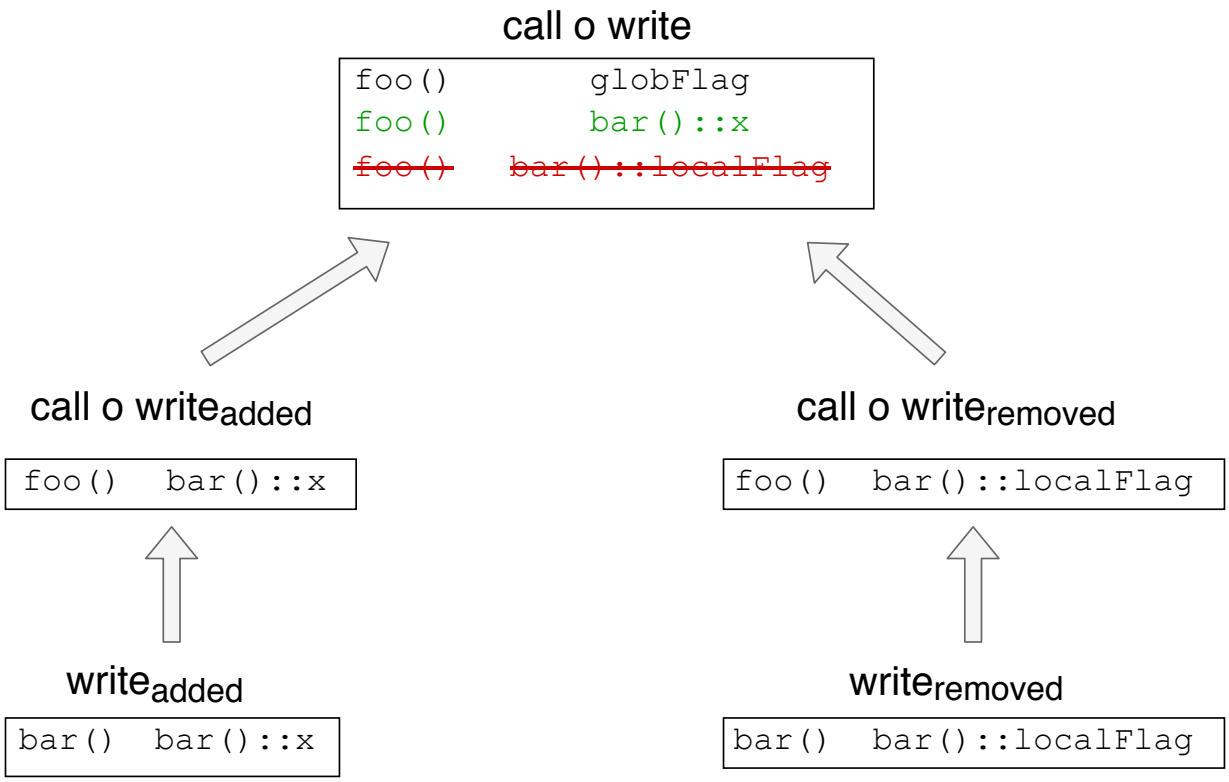


Figure 2.6: Updating a Result Set by Propagating Intermediate Results

Our work uses a similar, less memory-intensive approach, the difference being that we propagate expressions analogous to those seen in the relational model, which we call *delta queries*, that are used to update analysis results rather than propagating the intermediate results.

2.7 Summary

In this chapter, we described how a typical Fact Extraction and Analysis framework can be used to analyse models of software systems extracted from source code artifacts using a process similar to separate compilation. We then discussed alternative formats of representing these extracted software system models and how these formats have been used in the literature. Finally, we reviewed related approaches to updating software system models and approaches to updating analyses of software system models, the latter providing an overview of Incremental View Maintenance techniques for relational and graph data.

Chapter 3

Methodology

Separate Fact Extraction has been shown to scale well to large, industrial systems [35]. Using similar principles to the compilation process, we can create factbases using similar time and resources. This approach, however, is not ideally suited to support the evolutionary nature of software systems; changes in source code necessitate re-extracting a factbase from scratch and then redoing any analyses from scratch, regardless of the magnitude of the changes.

In this chapter, we outline an approach for making fact extraction more amenable to changes in the source code; we call this approach the Incremental Extraction and Analysis Process (IEAP). We first review the typical pipeline process for creating a system model from a set of source code files, which we call the Naive Extraction Process (NEP). The NEP is used the first time a system model of a given software system is created, producing intermediate artifacts along the way. Rather than apply the NEP every time a software system evolves, we propose an alternative approach that uses these intermediate artifacts and changes to source code files — namely modifications to source files, addition of new source files and removal of source files — to update a previously extracted system factbase; we elaborate on our approach to incremental extraction in Section 3.2. Next, we review the typical procedure for producing a higher-level model from an extracted system model, which we call the Naive Analysis Process (NAP). Similar to the NEP, the NAP can be applied on successive versions of the system model. We then propose an alternative approach that uses the changes made to a system model as it evolves to update the results of a previous analysis; we elaborate on our approach to incremental analysis in Section 3.4.

3.1 The Naive Extraction Process (NEP)

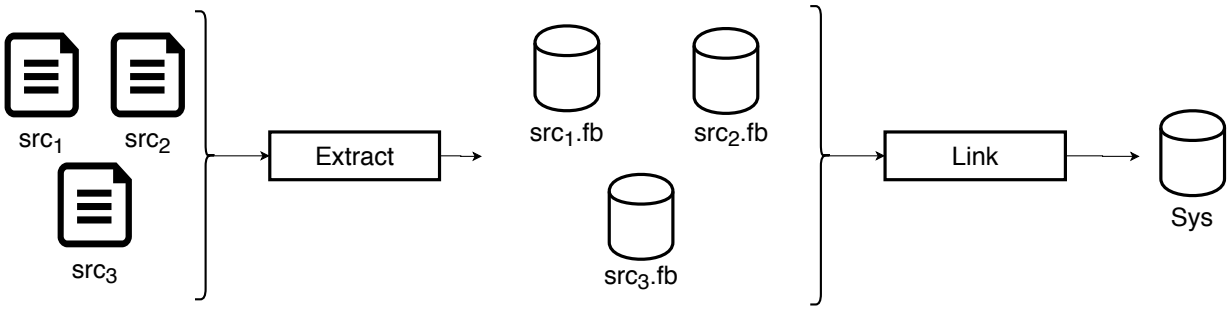


Figure 3.1: The Naive Extraction Process

As discussed earlier, the NEP in Figure 3.1 consists of two main phases:

1. *Separate extraction* — For each source code file in the software system src_i, a fact extractor is run on it to produce a preliminary, local factbase $src_i.fb$ containing the program entities that are defined in that file, such as functions and global variables, the relationships between such elements, such as function calls and variable writes, and additional properties/attributes for the extracted entities and relationships.

2. *Merging and linking* — Once the factbases of all of the source code files have been extracted, the individual fact bases are merged into a single factbase *Sys* that contains all of the facts for the system. A kind of linking is performed to resolve relations that span file boundaries, similar to how linkers perform linking on the object files of a program to resolve references to common identifiers (contained in the object files). Specifically, two types of resolutions among facts in disjoint factbases are performed:

   Name resolution — For each node/entity V in an extracted edge/relation E, we check that V has been extracted by the fact extractor. Nodes that are not extracted typically correspond to library functionality that can be ignored (e.g., calls to system libraries). If V has been extracted, then E is **resolved** and therefore included in *Sys*. If E is **unresolved** at the end of the process of linking all the local factbases, it is excluded from *Sys*.

   Property resolution — We merge all extracted nodes/entities with the same name/symbol $V_1 ... V_n$ into V_{Sys}, combining the separate, local-level properties into one system-level property using pre-defined *merge rules* (which are defined

per-property). Similarly, we merge all resolved edges/relations with the same name/symbol $E_1 ... E_n$ into E_{Sys}, combining the separate, local-level properties into one system-level property using pre-defined *merge rules* (which are defined per-property).

The NEP can be used on subsequent versions of a software system, however this necessitates reconstructing from scratch the individual factbases for any source code modules that changed and linking from scratch all the individual factbases.

3.2 The Incremental Extraction Process (IEP)

The Incremental Extraction Process (IEP) attempts to take advantage of the knowledge gains about a software system, i.e., which components have been added, modified, removed, etc. and perform as little additional work as possible to produce a correctly updated system model Sys' for the new version of the system. To explain how the IEP works, it is useful to observe that source code artifacts in a new version of the system — henceforth, referred to as *files* — can be partitioned into four categories:

An *Unchanged* file is present in both versions of the system, and is identical in both versions.

A *New* file is present in the new version of the system, but not in the old version.

A *Deleted* file is present in the old version of the system, but not in the new version.

A *Changed* file is present in both versions of the system, but is not identical in both versions.

We make the simplifying assumption that a file's identity corresponds to the full path name of the file within the source distribution directory hierarchy. If a file is renamed or moved to a different directory, we assume the old file was *Deleted* and the new file is *New*.[1] A file is considered to be *Unchanged* if the Unix `diff` command produces no output when using the old and new versions of the file as input.

[1]The problem of moving and renaming files between consecutive versions of a system is called origin analysis [20]; techniques such as code clone detection [40] can be used to recognize when this occurs. This can help to further optimize the IEP, but we consider it to be beyond the scope of our current work.

For each of these categories, there is a corresponding category of a file's corresponding local factbase: *Unchanged*, *New*, *Deleted* and *Changed*. A factbase's name is constructed using a file's full path name. Thus a file that is renamed or moved to a different directory causes the factbase associated with the old filename to be considered *Deleted* and the factbase associated with the new filename to be considered *New*. In addition, we introduce an additional artifact that is generated by both the NEP and the IEP: a factbase 'list'. Simply put, a factbase 'list' is a list of the names of the extracted local factbases that make up the system factbase, sorted in alphabetical order. Using a merge-like routine, we can then compare the old factbase 'list' to the new factbase 'list' and partition the factbases into the four aforementioned categories:

A factbase $src_i.fb$ is *Unchanged* when the factbase is extracted from both versions of the software system, and is identical in both versions.

A factbase $src_i.fb$ is *New* when the factbase is extracted from the new version of the software system, but not from the old version.

A factbase $src_i.fb$ is *Deleted* when the factbase is extracted from the old version of the software system, but not from the new version.

A factbase $src_i.fb$ is *Changed* when the factbase is extracted from both versions of the software system, but is not identical in both versions.

The IEP is broken down into the three major steps shown in Figure 3.2:

Re-Extract — For each *Changed* or *New* file src_i in the new version of the system, we re-extract the local factbase $src_i.fb.new$ using an approach similar to incremental compilation. We also generate the new factbase 'list' during this step. This process is explained in more detail in Section 3.2.1.

Diff and Replace — We detect which facts should be added, modified or removed from the system factbase *Sys* as a result of changes to the source code files, generating a file-level δ for each factbase category:

- For each *Unchanged* factbase $src_i.fb$, the resulting δ is null.
- For each *New* factbase $src_i.fb$, the resulting δ, $src_i.fb.diff$, contains only facts to be added to *Sys*, marked by a '+'.
- For each *Deleted* factbase $src_i.fb$, the resulting δ, $src_i.fb.diff$, contains only facts to be removed from *Sys*, marked by a '−'.
- For each *Changed* factbase $src_i.fb$, the resulting δ, $src_i.fb.diff$, contains facts to be added to *Sys*, marked by a '+', facts to be removed from *Sys*, marked by a '−', and fact attributes/properties to be modified in *Sys*.

We call this collection of file-level δ's a Δ. We also maintain any intermediate artifacts by replacing older versions of *Changed* factbases with their newer versions, removing *Deleted* factbases and replacing the old factbase 'list' with the new factbase 'list'. This process is explained in more detail in Section 3.2.2.

Update — We use our generated Δ to update our previously extracted system factbase, progressively applying each file-level $\delta \in \Delta$ to produce the updated system model Sys'. This process is explained in more detail in Section 3.2.3.

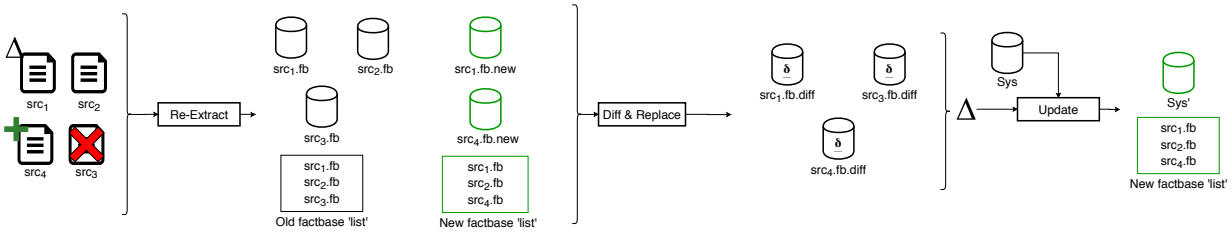


Figure 3.2: The Incremental Extraction Process

3.2.1 Re-Extracting

To minimize the number of source files we re-extract, we go through the following process. For each given source file src_i, we check to see if a corresponding factbase $src_i.fb$ exists. If the factbase does not exist, then we extract it and add an entry to the new factbase 'list'. Otherwise, we re-extract iff any of the following Re-Extraction conditions are satisfied:

- the source file has been written to since it was last extracted
- the source file's dependencies have changed since it was last extracted
- the source file's compilation command has changed since it was last extracted. This is applicable only when extracting from a built system

This information, namely a source file's last write time, a source file's dependency list and a source file's compilation command, is included in each local factbase $src_i.fb$ as metadata.

3.2.2 Diff & Replace

Following re-extraction, we need to know which local factbases have been added, removed and modified as a result of source code changes and, accordingly, which facts have been added, removed and modified. We do this by comparing the old factbase 'list' with the new factbase 'list'. Knowing that both lists are sorted alphabetically, we can detect whether a factbase has been removed, added or modified.

If a local factbase has been removed in the new extraction (i.e., a factbase that is present in the old factbase 'list' and absent in the new factbase 'list'), then we use the contents of the removed local factbase to generate a set of facts to be removed from the *Sys* factbase (denoted by a '$-$'), which we store in a file-level δ. If a local factbase has been added in the new extraction (i.e., a factbase that is absent in the old factbase 'list' and present in the new factbase 'list'), then we use the contents of the added local factbase to generate a set of facts to be added to the *Sys* factbase (denoted by a '$+$'), which we store in a file-level δ. If a local factbase has been modified (i.e., if there exists a newly extracted local factbase that is not deemed as *New*, such as $src_1.fb.new$ in Figure 3.2), then we compare the contents of the newly generated factbase with contents of the previous version of that local factbase, both of which are sorted, generating a set of facts to be added, removed and modified (at the attribute level) in the *Sys* factbase, which we store in a file-level δ. The collection of generated file-level δ's are collected in a Δ. For example in Figure 3.2, $\Delta = \{src_1.fb.diff,\ src_3.fb.diff,\ src_4.fb.diff\}$. A write-up of this algorithm can be found in Appendix A.1.

3.2.3 Update

The *instances* property

Recall that the linking step of Separate Fact Extraction includes the merging of same-named nodes (and edges) that appear disjoint in local factbases; when merging nodes (and edges), their properties are either merged or overridden. Depending on the semantics of a node's or edge's properties, tracing a system-level node or edge to its local-level constituents can be quite cumbersome.

To avoid a similar complication in our Update procedure, we have introduced an *instances* property for nodes and edges in the *Sys* factbase and in the local factbases. In a *Sys* factbase, this property represents the number of local factbases that contain the extracted node or edge, while in the local factbases, this counter represents the number of times a node or edge was extracted from a source file. For example, if function `foo()` calls `bar()` twice in `two.c` and once in `one.c`, then the fact `call foo() bar()` will have an *instances* value of 2 in `two.c.fb`, 1 in `one.c.fb` and 3 in the *Sys* factbase.

Applying Δ

Once we know which facts have been added, removed and modified, we need to produce a revised *Sys* factbase that includes these changes. We apply our update in two parts. First, we iterate over each file-level $\delta \in \Delta$ and apply any node, edge or property removals. Removing local factbase properties from the *Sys* factbase requires background knowledge on the semantics of the properties. For example, the *instances* property is a counter; hence, if an *instances* property is removed from a local factbase then our Update process decrements the property's value in *Sys* by the property's value in the local factbase. If the number of *instances* of an edge drops to zero, then the edge is removed from the *Sys* factbase. If the number *instances* of a node drops to zero, then both the node and any edges connecting to that node are removed from the *Sys* factbase. Other possible property semantics include the *condition* property, which is a boolean expression denoting a fact's *presence condition* and is used when lifting analyses to software product lines [43], and the *filename* property, which denotes the source file from which a fact was extracted. A write-up of this algorithm can be found in Appendix A.2

Next, we iterate over each file-level $\delta \in \Delta$ a second time and apply any node, edge or property additions and modifications. As with removing properties from *Sys* factbase, adding or modifying properties in the *Sys* factbase based on added or modified properties from a local factbase requires knowing the semantics of these properties. As well, we need

to allow for the fact that new edges (relations) may refer to nodes that have not yet been added. For example, a function call may be extracted from one file, and the called function may be declared in another file. In such a case, our Update process creates *virtual nodes* that serve as placeholders for nodes that are to be added. Should these placeholders not be filled by the end of the Update process, then they are removed along with any outgoing and incoming edges. We identify *virtual nodes* as nodes that do not have the *instances* property. A write-up of this algorithm can be found in Appendix A.2

3.2.4 Creating *Meta Facts*

With a view to supporting Incremental Analyses, the IEP generates the changes made to the *Sys* factbase during the *Update* step, partitioning it into two categories:

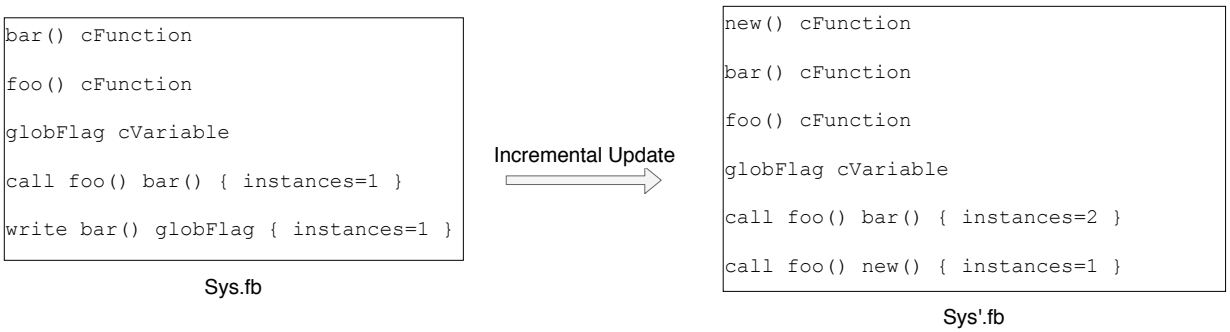
> *Dirty Facts* — These are facts that have been modified or added by the IEP when updating *Sys* to *Sys′*, the term *Dirty* being borrowed from the notion of a dirty bit. For each fact $f_i^{Sys'}$ that has been modified or added, a copy f_i^{Dirty} of the fact is created whose `dirty` attribute is set to `true`. Each *Dirty Fact* f_i^{Dirty} is treated as a *New* fact in the IAP, even if the fact was present in the old system model.
>
> *Removed Facts* — These are facts that have been removed by the IEP when updating *Sys* to *Sys′*. For each fact f_i^{Sys} that is removed, a copy $f_i^{Removed}$ of the fact is created whose `removed` attribute is set to `true`.

We call this collection of *Dirty* and *Removed* facts *Meta Facts*; these *Meta Facts* constitute the changes to the software system factbase ΔSys. Consider Figure 3.3 as an example. When *Sys.fb* is updated to *Sys′.fb* in Figure 3.3a, the fact `call foo() bar()` is modified, the fact `call foo() new()` is added, and the fact `write bar() globFlag` is removed. Accordingly, the IEP generates the following *Dirty Facts* in Figure 3.3b:

```
call foo() bar() { dirty=true instances=2 }

call foo() new() { dirty=true instances=1 }
```

and this *Removed Fact*:

```
write bar() globFlag { removed=true }
```



(a) Incrementally Updating Factbases

```
write bar() globFlag { removed=true }

call foo() bar() { dirty=true instances=2 }

call foo() new() { dirty=true instances=1 }
```

(b) *Meta Facts* generated for the Incremental Update

Figure 3.3: Generating *Meta Facts* during an Incremental Update

3.3 The Naive Analysis Process (NAP)

The Naive Analysis Process (NAP) in Figure 3.4 consists of only one phase: *Analysis*. During this phase, we apply analyses to the extracted system model *Sys* to produce a set of analysis results called *Analysis(Sys)*. Such analyses may involve excluding facts that are not relevant to the problem at hand, inferring higher-level facts about the system, such as dependencies between components of the software system, or finding interesting patterns or smells (e.g., feature interaction hotspots). Together, *Analysis(Sys)* and *Sys* form a richer factbase of derived and extracted facts, respectively. The NAP must be used to produce the first set of analysis results from the first *Sys* model. When the *Sys* model evolves, we could use the NAP to re-generate up-to-date *Analysis(Sys)* results, however this is a lot of rework every time the software changes.

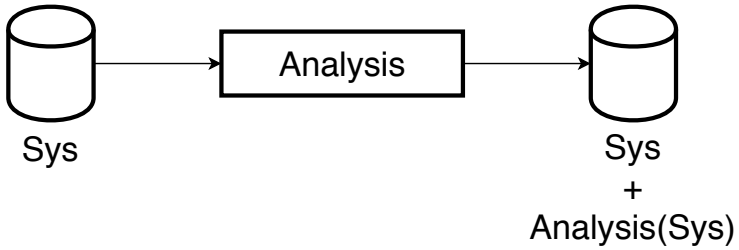


Figure 3.4: The Naive Analysis Process

3.4 The Incremental Analysis Process (IAP)

The Incremental Analysis Process (IAP) aims to use the changes to a software system factbase, referred to as ΔSys, to correctly update the results from a previous *Analysis(Sys)*, rather than perform the analysis from scratch on Sys'. As described in Section 3.2.4, this ΔSys is generated during the *Update* step of the IEP as *Meta Facts*.

The IAP is broken down into three steps, as depicted in Figure 3.5:

Detect Meta Facts — For each binary relation R_i in an analysis query *Analysis*, we identify the relevant subset of *Meta Facts* $f_1^{Meta}...f_j^{Meta}$ that are instances of R_i. This process is explained in more detail in Section 3.4.1.

Propagate Meta Facts — For each identified *Meta Fact* f_j^{Meta}, we generate an expression Q_j, which we call a *delta query*, using the syntactic structure of an analysis

query *Analysis*. Conceptually, if f_j^{Meta} is a *Dirty Fact* then Q_j is an *additive delta query*, which we denote as Q_j^+; it represents the set of additions to be made to *Analysis(Sys)* due to f_j^{Meta}. Likewise, if f_j^{Meta} is a *Removed Fact* then Q_j is a *removal delta query*, which we denote as Q_j^-; it represents the set of removals to be made *Analysis(Sys)* due to f_j^{Meta}. This process is explained in more detail in Section 3.4.2.

Update Analysis(Sys) — We evaluate our set of *delta queries* $Q_1 ... Q_k$ to generate facts that are added to *Analysis(Sys)* and identify facts that are removed from *Analysis(Sys)*. These *delta queries* are performed on the updated system model *Sys′*. This process is explained in more detail in Section 3.4.3.

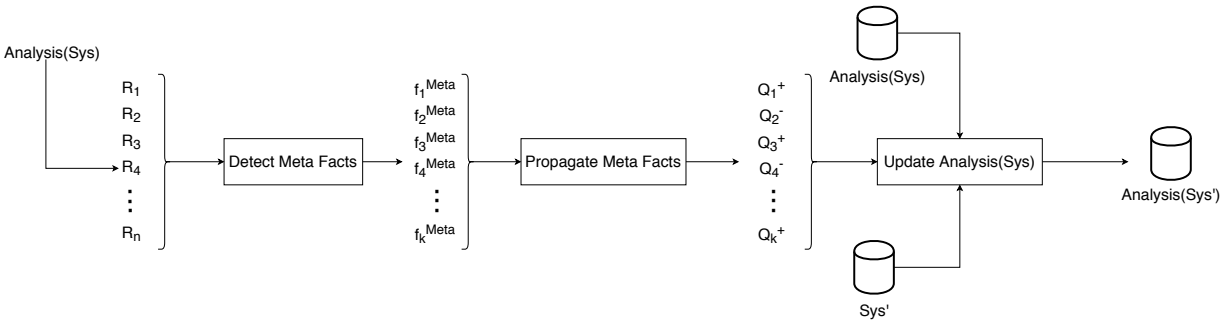


Figure 3.5: The Incremental Analysis Process

3.4.1 Detecting *Meta Facts*

The first step in the IAP is to identify facts that are relevant to a particular analysis query and have either been removed or added. For example, suppose the analysis query *call* ∘ *write* was executed on *Sys.fb* in Figure 3.3a. When *Sys.fb* is updated to *Sys′.fb*, the IAP must identify:

- The removed fact `write bar() globFlag`
- The added fact `call foo() new()`

Our approach does this by mapping the relations used in the analysis query to relevant *Meta Facts*. For the analysis query *call* ∘ *write*, the IAP identifies from the set of *Meta Facts* in Figure 3.3b, which in this case happens to be the entire set of *Meta Facts*:

- The *Removed Fact* `write bar() globFlag`
- The *Dirty Fact* `call foo() new()`
- The *Dirty Fact* `call foo() bar()`

Note that the identified *Dirty Fact* `call foo() bar()` is redundant in the context of the analysis query $call \circ write$, as the fact `call foo() bar()` was not added to $Sys'.fb$, the consequence being a negative impact on performance. Identifying redundant *Meta Facts* can be avoided if a history of facts with respect to a given analysis is maintained, however we consider it to be beyond the scope of our current work.

3.4.2 Propagating *Meta Facts*

Once we determine which facts relevant to an analysis have been added and removed, we need to perform some inference to determine how *Analysis(Sys)* should be updated based on these changes. To achieve this, the third step of the IAP propagates the detected changes in a manner that is similar to a Rete network [17], generating *delta queries* that it uses to:

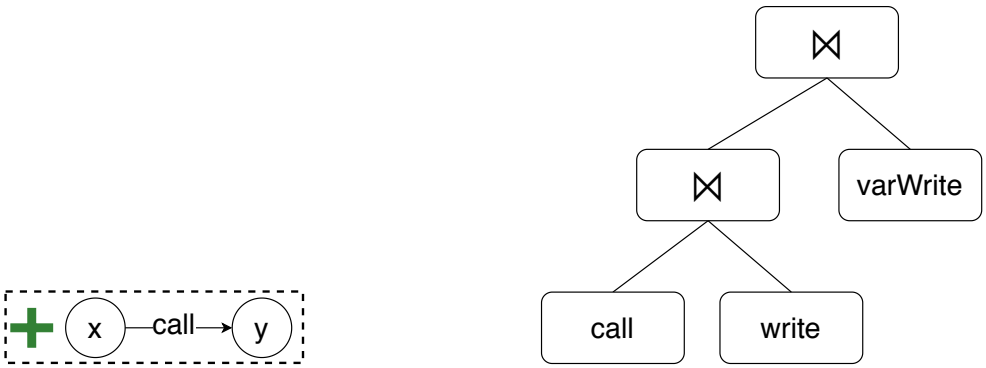
- yield a set of analysis results to be added to *Analysis(Sys)*
- yield a set of analysis results to be removed from *Analysis(Sys)*

Suppose you want to run the analysis: $A \mathrel{\widehat{=}} call \bowtie write \bowtie varWrite$, where:

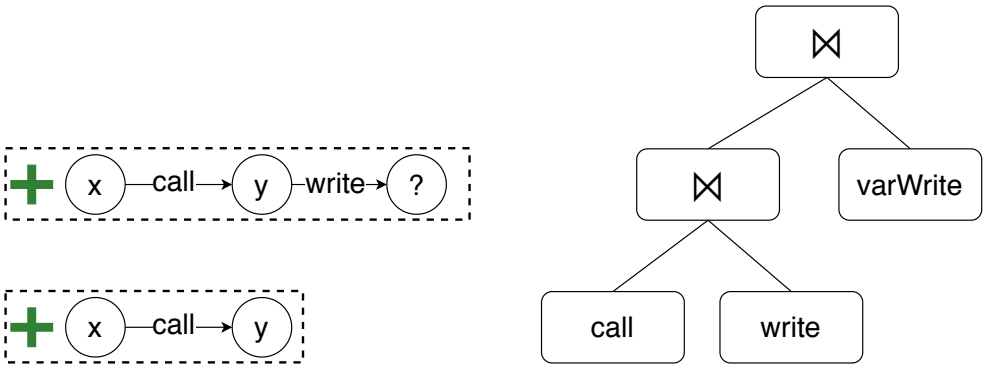
> The *call* relation represents the set of all ordered pairs *(a,b)* where a and b are functions and a directly calls b
>
> The *write* relation represents the set of all ordered pairs *(b,c)* where b is a function, c is a variable and b directly writes to c through an assignment
>
> The *varWrite* relation represents the set of all ordered pairs *(c,d)* where c and d are variables and c writes its value to d through an assignment

A therefore identifies functions that affect the value of a variable assignment through a function call. We can represent this query as a tree, where the leaves are the *call*, *write* and *varWrite* relations and the inner nodes are the join ($\bowtie$) operators, including the root.

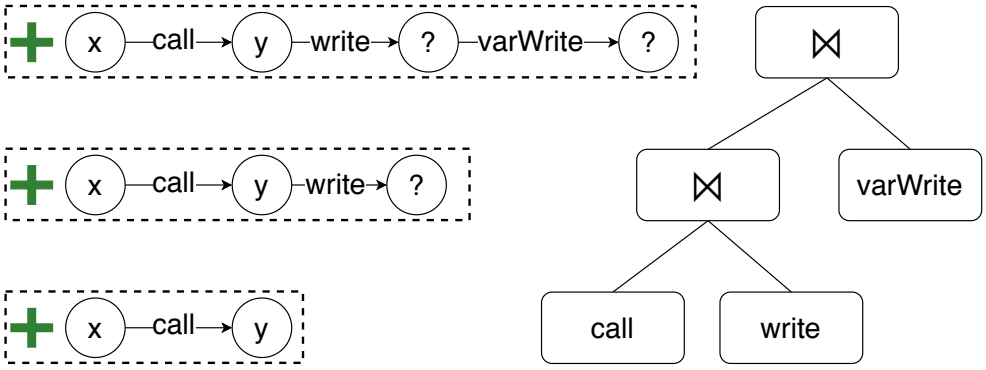
We can then use the leaves to detect relevant *Meta Facts*. Consider the example in Figure 3.6a, where the leaf node **call** was used to find a *Meta Fact* stating that the call from `x` to `y` is dirty. We interpret dirty facts as being additions, and so the call from `x` to `y` is considered to be a new fact to be added. This change is propagated up one level to Figure 3.6b, generating the *delta query* on the left side of the Figure. At this stage, this *delta query* represents the set of intermediate results to be added to *Analysis(Sys)* based on the call from `x` to `y` being added. When the change is propagated further to the root in Figure 3.6c, the final *delta query* on the left represents the set of all new additions that should be made to *Analysis(Sys)* based on the call from `x` to `y` being added. Propagating detected removals follows the same procedure, the only difference being that the pattern at the root now represents the set of all possible facts to be removed from *Analysis(Sys)*.



(a)



(b)



(c)

Figure 3.6: Propagating a Dirty Fact

3.4.3 Updating *Analysis(Sys)*

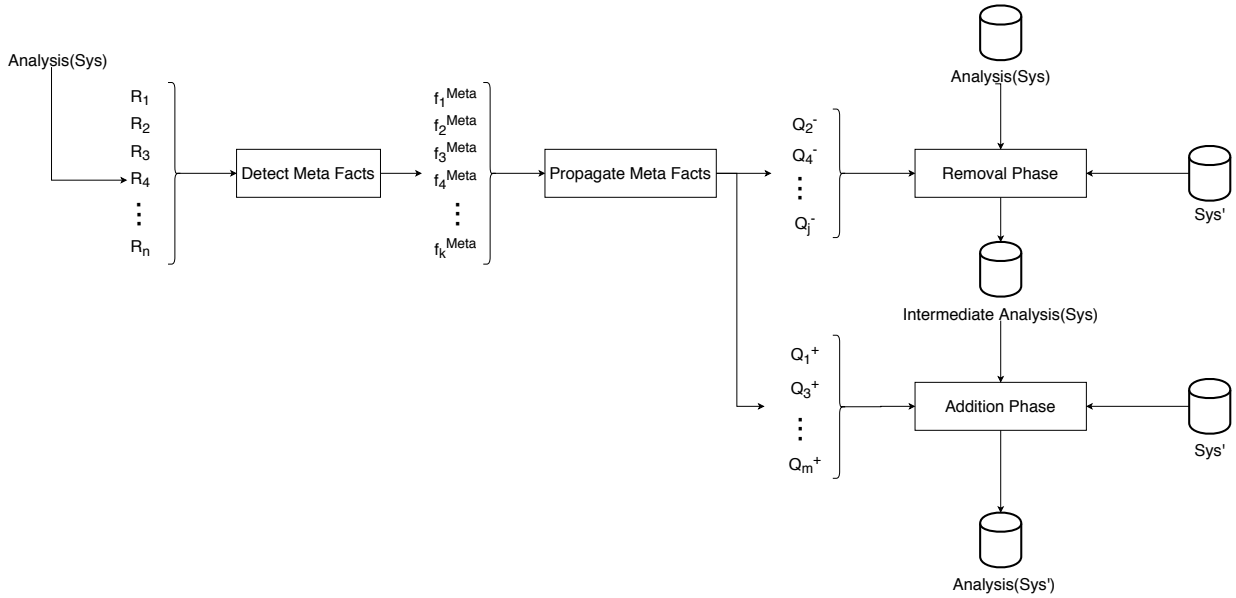


Figure 3.7: The *Update* stage of the IAP

The final step in the IAP updates *Analysis(Sys)* by evaluating our generated *delta queries* on the updated system factbase *Sys′*. This approach is split in two stages, as shown in Figure 3.7: the removal phase and the addition phase.

The Removal Phase

During the removal phase, we need to remove results from *Analysis(Sys)* that can no longer be derived in the new system factbase *Sys′*. The first step to identifying such results is to evaluate our generated set of *removal delta queries* $Q_2^-...Q_j^-$. These queries are incremental versions of the original analysis query *Analysis* that look for analysis paths that include their respective fact that was removed from *Sys* and is absent from *Sys′* ($f_2^{Removed}...f_j^{Removed}$). Since these queries inherently derive analysis results from facts that have already been removed from *Sys′*, evaluating $Q_2^-...Q_j^-$ on *Sys′* alone yields the empty set. Accordingly, the IAP creates for each of the removed facts $f_2^{Removed}...f_j^{Removed}$ dummy copies of the nodes and edges, producing the factbase $Sys'+Sys_{dummy}$. Executing $Q_2^-...Q_j^-$

on $Sys'+Sys_{dummy}$ now yields an initial set of facts $Removed_{initial}$ to be removed from *Analysis(Sys)*, which the IAP does in parallel for added efficiency.

Having identified an initial set of facts $Removed_{initial}$ to remove from *Analysis(Sys)*, the IAP removes all dummy nodes and edges from $Sys'+Sys_{dummy}$, producing Sys'. A result in $Removed_{initial}$ could still possibly be derived from Sys', however, as the *removal delta queries* look for analysis paths that include removed facts. It is possible that the same analysis result can be derived from an analysis of Sys' that does not include the removed facts. Hence, to prevent over-deleting from *Analysis(Sys')*, we need to check if each result in $Removed_{initial}$ can be derived from Sys'. The IAP does this by executing a localised version of the analysis query *Analysis* on Sys', where the localised query looks for an analysis path that includes the analysis result. For example, if `foo() bar()` is in $Removed_{initial}$ for the transitive closure of the *call* relation, the IAP would check if `foo()` transitively reaches `bar()` through the *call* relation in Sys'. All elements in $Removed_{initial}$ that can be derived from Sys' are purged from $Removed_{initial}$ by the IAP, since their inclusion would constitute an over-deletion. Once the purge is complete, the IAP removes any remaining elements in $Removed_{initial}$ from *Analysis(Sys)*.

The Addition Phase

The addition phase is relatively simpler than the removal phase. The IAP need simply execute our additive *delta queries* $Q_1^+...Q_m^+$, producing a set of results to be unioned with the *Analysis(Sys)*. For added efficiency, these queries can be executed in parallel.

3.5 Implementing the IEAP

Having defined our Incremental Extraction and Analysis Process (IEAP), we put it into practice by modifying the extraction and analysis toolchain laid out in [35], which followed the Naive Extraction and Analysis Process. We chose Neo4j 3.5.6[2] as our backend rather than TA/Grok, which was used in the original toolchain, as we wanted the IEAP to be applicable with an industrial-scale graph database system. We implemented the IEP by first modifying the toolchain's fact extractor *Rex*[3] to:

1. Generate factbase 'lists'
2. Generate file-level δ's for local factbases

This allowed *Rex* to carry out the *Re-Extract* and *Diff and Replace* stages of the IEP. We implemented the IEP's *Update* procedure as a separate program using Neo4j's Native Java API[4], completing our implementation of the IEP. Similarly, we developed a prototype of the IAP using Neo4j drivers for C[5]. All of our analyses are executed using APOC's `subgraphNodes()` function in the `pathExpander` library[6].

3.6 Summary

In this chapter, we introduced the Incremental Extraction and Analysis Process (IEAP), a modification of the Separate Extraction framework. As a software system evolves, the IEAP leverages changes to source code files to update previously extracted system models of the software system and previous analyses/post-processes of the system model. We also outline an implementation of the IEAP using an existing extraction and analysis toolchain, noting the modifications made to the toolchain to incorporate the IEAP.

[2] https://neo4j.com
[3] https://git.uwaterloo.ca/swag/Rex: `Rex-incremental` branch
[4] https://git.uwaterloo.ca/swag/incremental-factbase-updater
[5] https://git.uwaterloo.ca/swag/incremental-fact-analyser
[6] https://neo4j.com/docs/labs/apoc/current/graph-querying/path-expander

Chapter 4

Case Study

In this chapter, we discuss a case study we performed using an industrial-scale open source system (the Linux Kernel) as the target. In parituclar, we examined three aspects of our proposed approach:

1. We compared the speed and accuracy of the IEP against the NEP,
2. We compared the speed and accuracy of the IAP against the NAP,
3. We identified sources of bottlenecks in the IAP.

We now describe our results in detail.

4.1 The Linux Kernel as the Target System

We wanted to evaluate the effectiveness of our incremental processes on an industrial-scale software system. We selected the Linux Kernel as the target system: it is open source and thus available to be used as a target; it is a large, long-lived and industrially successful system that is in very wide use; and it is written in the C language, which our toolset supports. Table 4.2 shows the size of the system per version, which was computed using the `cloc` utility, and the size of the extracted factbases per version. In summary, our corpus had medians of 200,063 SLOC[1], 432 source files, 115,856 nodes, and 243,296 edges.

[1]The raw size of the Linux kernel is over 15M SLOC; we are using an "allNoConfig" option with minimal add-ons, and the size of this subset is about 200,000 KSLOC

The nodes consisted of the following program elements: variables, functions, enumerations. The edges consisted of the relations in Table 4.1 below.

Table 4.1: Extracted Relations from Linux Corpus

Relation	**Meaning**
`call` *function1 function2*	*function1* calls *function2*
`write` *function variable*	*function* assigns data to *variable*
`read` *variable function*	*function* reads data from *variable*
`varWrite` *variable1 variable2*	*variable1* assigns its data to *variable2*
`contain` *entity1 entity2*	*entity1* contains *entity2*; e.g., a function contains a local variable
`varInfFunc` *variable function*	*variable* is used in a control-flow statement that affects whether *function* is called

Table 4.2: Linux Corpus

Version	*Source Size*		*Factbase Size*	
	# of Src Files	*SLOC*	*# of Nodes*	*# of Edges*
v4.0	416	177,261	94,881	201,233
v4.1	417	181,046	96,398	204,526
v4.2	421	183,995	98,835	208,519
v4.3	424	185,455	100,037	211,038
v4.4	426	186,358	102,373	213,741
v4.5	426	188,299	103,934	216,642
v4.6	428	190,958	105,612	220,576
v4.7	429	191,966	106,284	222,086
v4.8	430	194,804	108,484	226,557
v4.9	432	196,736	110,222	231,302
v4.10	430	196,279	110,412	230,857
v4.11	431	197,260	112,099	234,847
v4.12	432	198,777	112,687	236,464
v4.13	431	200,063	115,856	243,296
v4.14	431	201,572	116,059	245,367
v4.15	435	203,569	116,860	247,444
v4.16	436	205,009	117,835	249,250
v4.17	438	207,103	119,580	252,666
v4.18	438	207,071	120,546	253,825
v4.19	438	208,139	121,556	256,000
v4.20	441	210,281	122,132	256,560
v5.0	443	211,292	123,182	258,607
v5.1	444	214,311	124,847	262,358
v5.2	445	216,878	126,415	266,095
v5.3	447	219,333	128,763	269,166
v5.4	448	221,300	129,921	271,667
v5.5	449	221,825	130,417	272,514

4.1.1 Setup

Building & Extracting 'allNoConfig' Linux

To run an extraction on a particular release of Linux, we checked out the release from its GitHub repository[2] using its release tag. We then compiled the release into an 'allNoConfig' build by selecting "no" to all the configuration options. Since Rex uses compilation databases in its extraction, we used the BEAR tool[3] to generate a compilation database per build. Each compilation database contains the path names of all the source code files that make up a build with their respective compiler flags.

Updating Extractions

To apply the NEP, we ran Rex in a batch-extraction mode on each of the Linux releases, generating for each release two CSV files[4]: *nodes.csv*, which contained all the extracted nodes/entities and their properties/attributes, and *edges.csv*, which contained all the extracted edges/relationships and their properties/attributes. We recorded the time taken to produce these CSV files per release, but did not record the time taken to bulk import the CSVs into Neo4j using the `neo4j-admin import` tooling as we felt that the extraction was complete once the up-to-date versions of the CSVs had been produced.

To apply the IEP, we ran Rex in its incremental-extraction mode on the sequence of Linux releases followed by our implementation of the IEP's *Update* procedure, applying the results of the first incremental extraction (version 4.1) to the results of the batch extraction of version 4.0 (after importing the generated CSVs into Neo4j), and thereafter applying the results of an incremental extraction of a version v_i to the factbase associated with the previous version v_{i-1}. We recorded the time it took for Rex to run in its incremental-extraction mode and the time taken by our Incremental Update procedure.

4.1.2 System Specifications

All experiments were performed on a machine running Ubuntu 16.04 LTS with an Intel Core i7-6770HQ 2.60 GHz CPU and 16GB RAM.

[2] https://github.com/torvalds/linux
[3] https://github.com/rizsotto/Bear
[4] A *Comma-Separated Values* (CSV) file is a text file in which values are separated by commas

4.1.3 Results

Figure 4.1 shows a high-level comparison of the speed of an Incremental Extraction with the speed of a Batch Extraction for successive versions of our target system. On initial viewing, we can see that the Incremental Extraction is at least 50% quicker than the Batch Extraction in all the updates, ranging from 54% to 65%. We validated the factabses produced when the two approaches (batch and incremental extraction) are applied to the same Linux update, finding that they were identical.

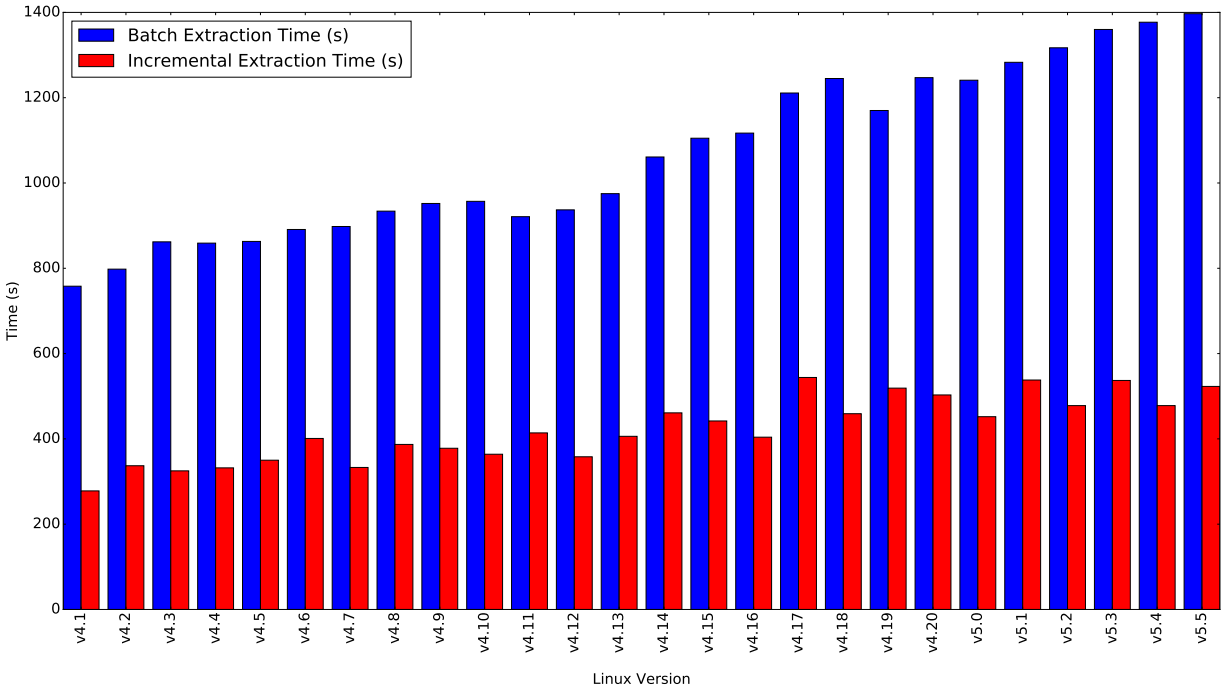


Figure 4.1: Incremental Extraction Compared With Batch Extraction

Both the Incremental and Batch Extraction use the same 'compile' step, where we follow a similar approach to Unix's *Make* utility to ensure we only re-extract facts from files that have changed since the last extraction (See Section 3.2.1), generating the same local factbases. Hence the only real gain that can be made is during the 'link' step. The magnitude of this gain is shown in Figure 4.2 where we can see a significant improvement in the 'link' step.

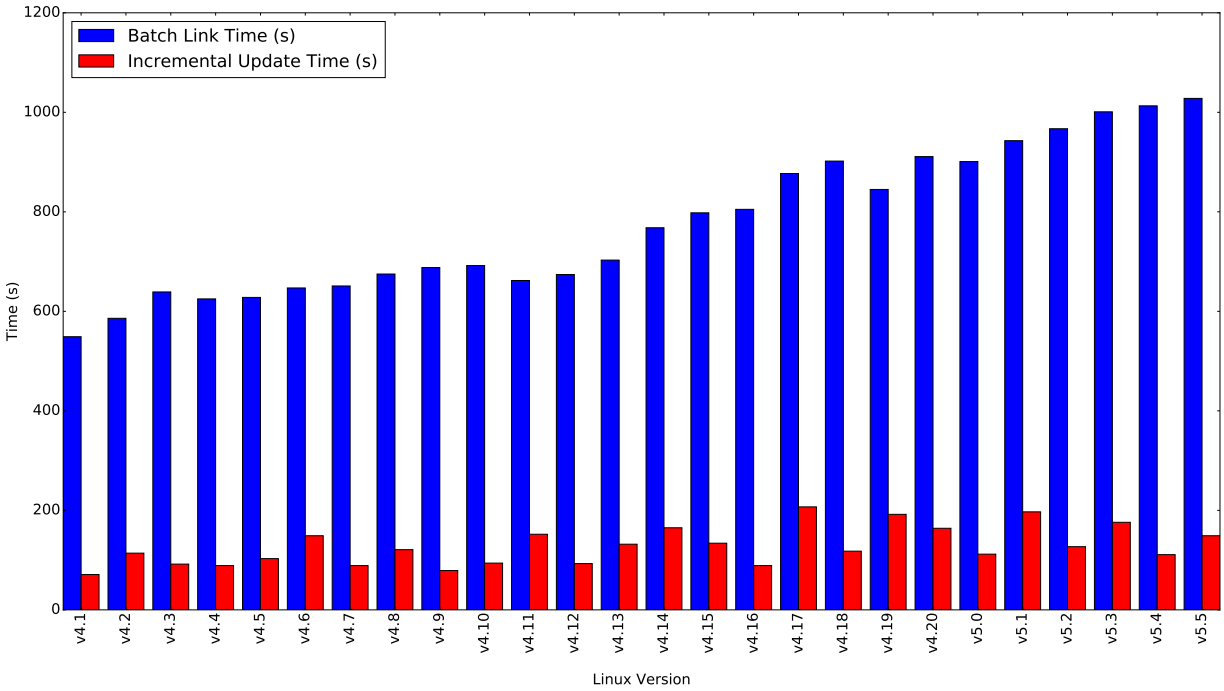


Figure 4.2: Incremental Updated Compared With Batch Link

While this improvement is encouraging, it is highly sensitive to the size of the collection of file-level δ's generated by the IEP, referred to as Δ. To that end, we explored how our Incremental Update scales with respect to the size of Δ, approximating this size by counting the number of facts (edges) and entities (nodes) in each partition of the file-level δ's in Algorithms 2 and 3. The correlation between the Incremental Update time and the size of Δ is shown in Figure 4.3, which on initial viewing seems to be linear. We measured the strength of this linearity using the Pearson correlation coefficient r, obtaining a value of 0.924. This indicates a strong, positive linear correlation between the time taken by our Incremental Update procedure and the size of Δ, which suggests that our Incremental Update procedure scales linearly with respect to the size of a Δ.

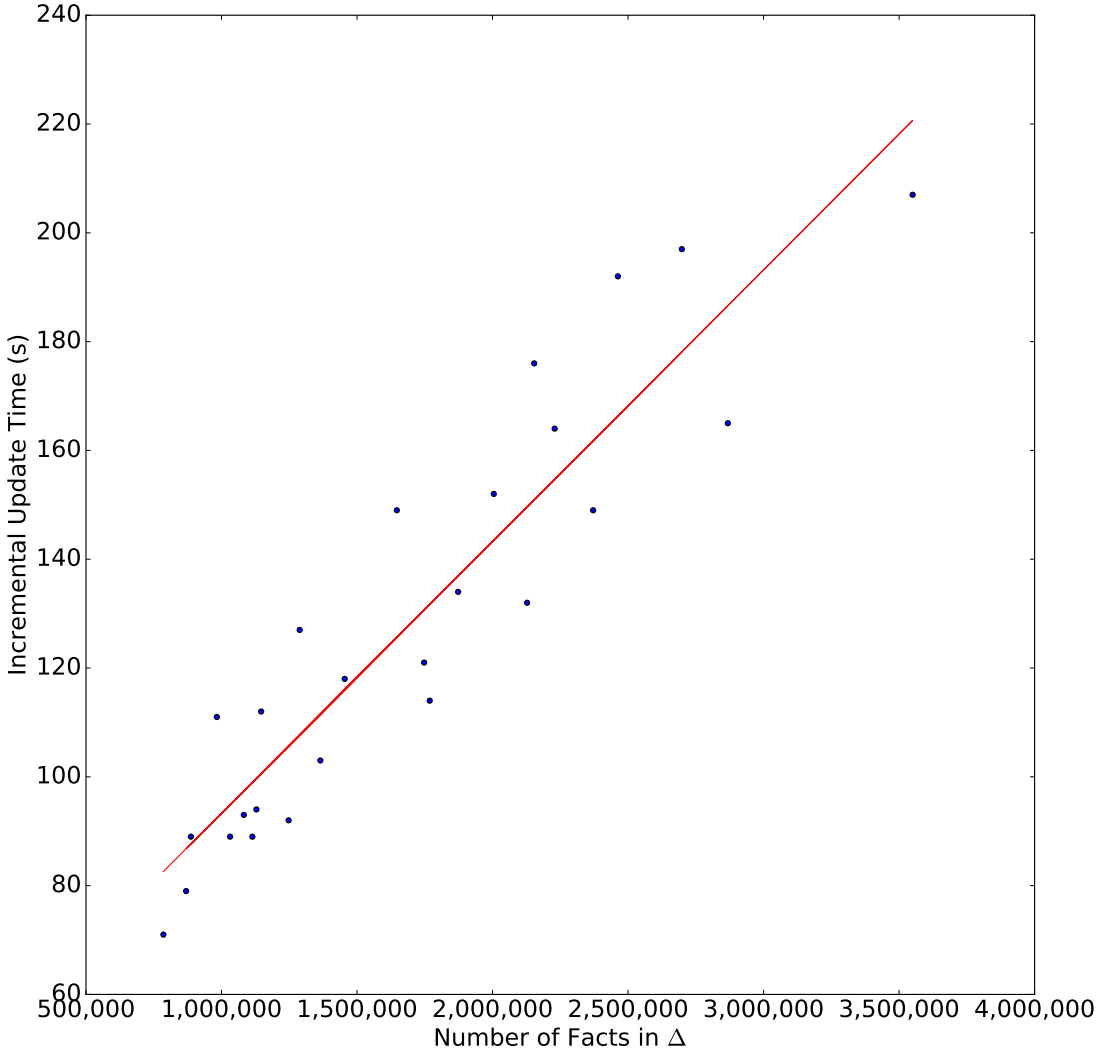


Figure 4.3: Growth of Incremental Update Time

4.2 Comparing the IAP to the NAP

Now that we have compared the IEP with the NEP, we compare the next part of the pipeline: the analyses.

4.2.1 Setup

First we extracted a System Factbase of an allNoConfig build of Linux v5.0, *linux.fb*, and ran an analysis query on the factbase, creating a result set *Analysis(linux.fb)*. We then ran an Incremental Extraction on an allNoConfig build of Linux v5.1 without performing an Incremental Update, generating only the collection of file-level δ's, Δ. We then selected increasingly larger subsets of Δ to create a variety of updated *linux.fb′*, each reflecting a larger increment of the initial *linux.fb* than the previous update. Each subset Δ_n of Δ consisted of the n smallest file-level δ's in Δ, meaning that each subset Δ_n is a superset of the previous subset Δ_{n-1}. Finally, we applied both the IAP (Incremental Analysis) and the NAP (Batch Analysis) to perform three distinct queries on each of the updated *linux.fb′* factbases, collecting performance data. Note that our Batch Analysis times include the time taken to remove the initial result set *Analysis(linux.fb)*.

The Analysis queries we used were:

$$call+ \tag{4.1}$$

$$call \bowtie (write \cup varWrite)+ \tag{4.2}$$

$$call \bowtie write \bowtie varWrite \tag{4.3}$$

where:

The *call* relation represents the set of all ordered pairs (a, b) where a and b are functions and a directly calls b

The *write* relation represents the set of all ordered pairs (b, c) where b is a function, c is a variable and b directly write to c through an assignment

The *varWrite* relation represents the set of all ordered pairs (c, d) where c and d are variables and c writes its value to d through an assignment

The '+' operator represents transitive closure

The '$\bowtie$' operator represents the natural join operation

Hence, Query 4.1 is just the transitive closure of the *call* relation, finding the set of all possible tuples (a, b) where a and b are functions and a either directly or indirectly calls b. Query 4.2 looks for all possible triples (a, b, c) where a is a function that calls a function b, and b assigns a value that directly or indirectly affects the value assigned to a variable c. Finally, Query 4.3 looks for all possible quadruples (a, b, c, d) where a is a function that calls a function b, and b assigns a value to a variable c, and the value of c is used in an assignment expression to variable d.

We chose Query 4.1 to see how well the IAP performs for a Query with a large search-space, while we chose Queries 4.2 and 4.3 to see how well the IAP performs for queries with increasingly smaller search spaces. Queries 4.1 and 4.2 both use transitive closure, which can expand an analysis' search space, and hence computation, exponentially. Query 4.3, in this respect, was chosen to contrast the IAP's performance with a query that does not use transitive closure, but is simultaneously non-trivial.

4.2.2 System Specifications

All experiments involving Queries 4.2 and 4.3 were performed on a machine running Ubuntu 16.04 LTS with an Intel Core i7-6770HQ 2.60 GHz CPU and 16GB RAM, while all experiments involving Query 4.1 were performed on a machine running Ubuntu 18.04 LTS with an Intel Xeon e5-1620 3.60 GHz CPU and 64GB RAM.

4.2.3 Results

Figures 4.4, 4.5 and 4.6 compare the speeds of an Incremental Analysis against the speeds of a Batch Analysis, when applied to increasing sizes of Δ. On initial viewing, we can see that an Incremental Analysis executes quicker than a Batch Analysis in the case of smaller Δ's. However for larger Δ's, the Incremental Analysis performs much worse than a Batch Analysis. We validated that when our Incremental and Batch analyses are performed on the same subset of Δ (used to produce an updated *linux.fb′*), they produce the same results *Analysis(linux.fb′)*.

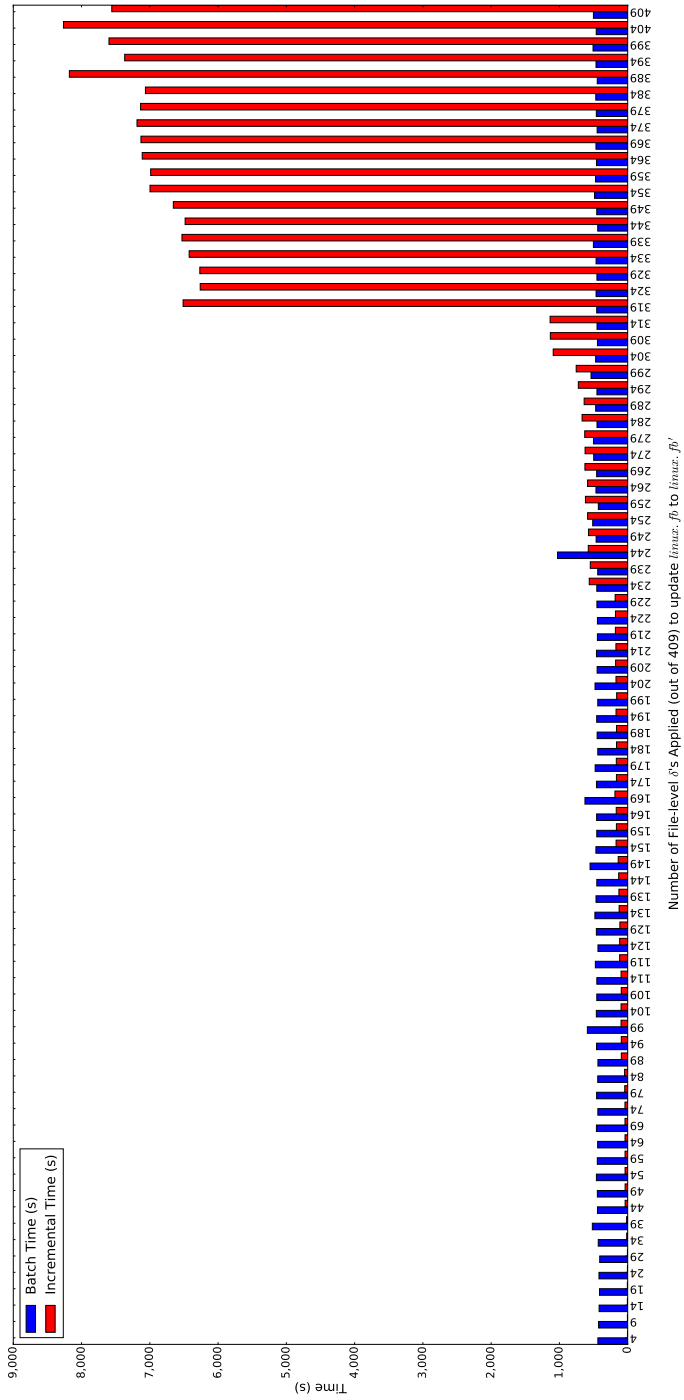


Figure 4.4: Incremental vs Batch Analysis on *call+*

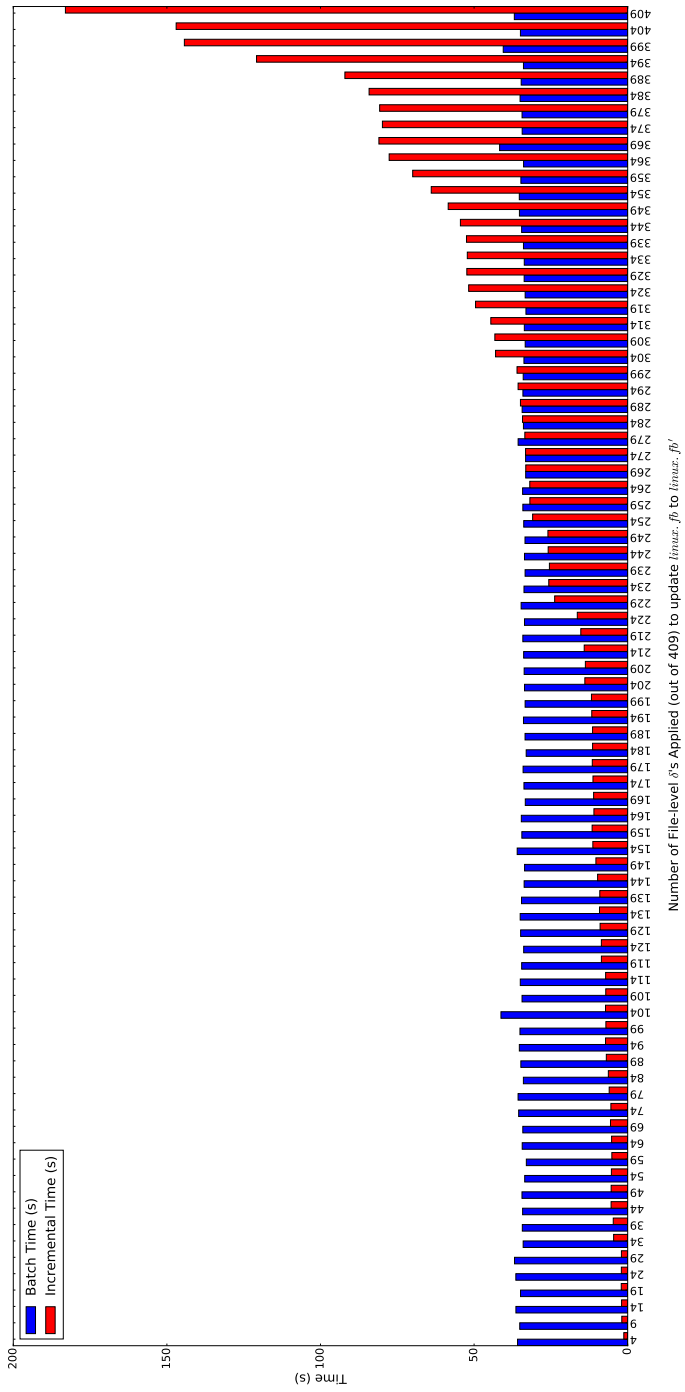


Figure 4.5: Incremental vs Batch Analysis on $call \bowtie (urite \cup varWrite)+$

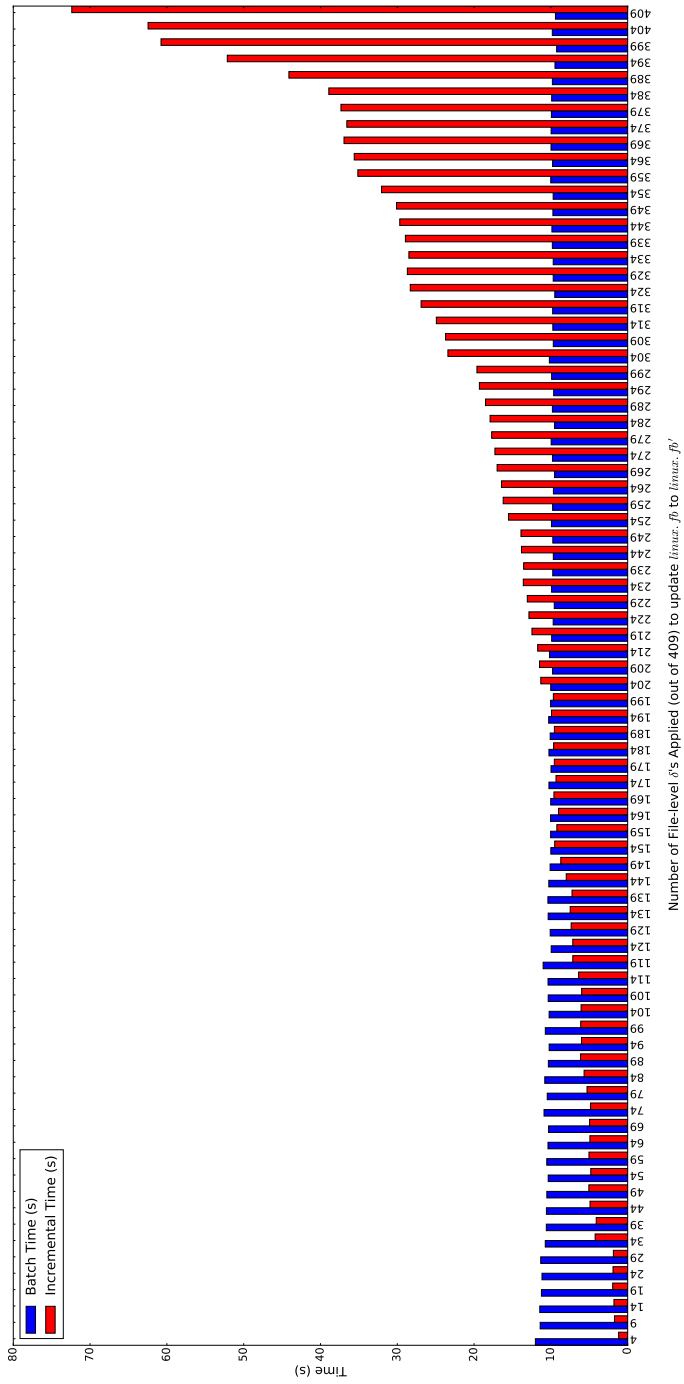


Figure 4.6: Incremental vs Batch Analysis on $call \bowtie write \bowtie varWrite$

To better understand this degradation in performance, we investigated the correlation between the execution time for an Incremental Analysis and the number of affected facts it has to process. In the cases of Queries 4.3 and 4.2, the number of affected facts is the number of *call*, *varWrite* or *write* relations that are either dirty or have been removed, while for Query 4.1 the number of affected facts is the number of *call* relations that are either dirty or have been removed. Intuitively, an Incremental Analysis (IAP) ought to scale linearly with respect to the number of affected facts, which is shown to be true for Queries 4.3 and 4.2 in Figure 4.9 and 4.8 respectively, however Figure 4.7 shows an S-shaped growth for Query 4.1; IAP initially scales linearly before growing exponentially, after which it stabilizes.

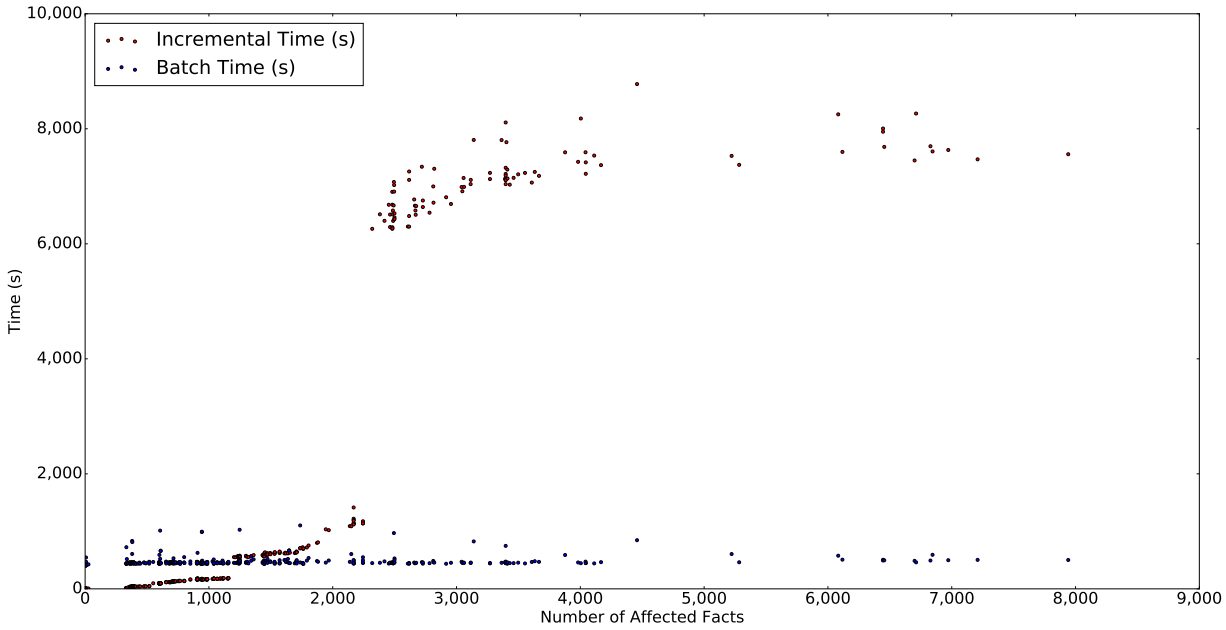


Figure 4.7: Incremental Analysis Growth for Query 4.1

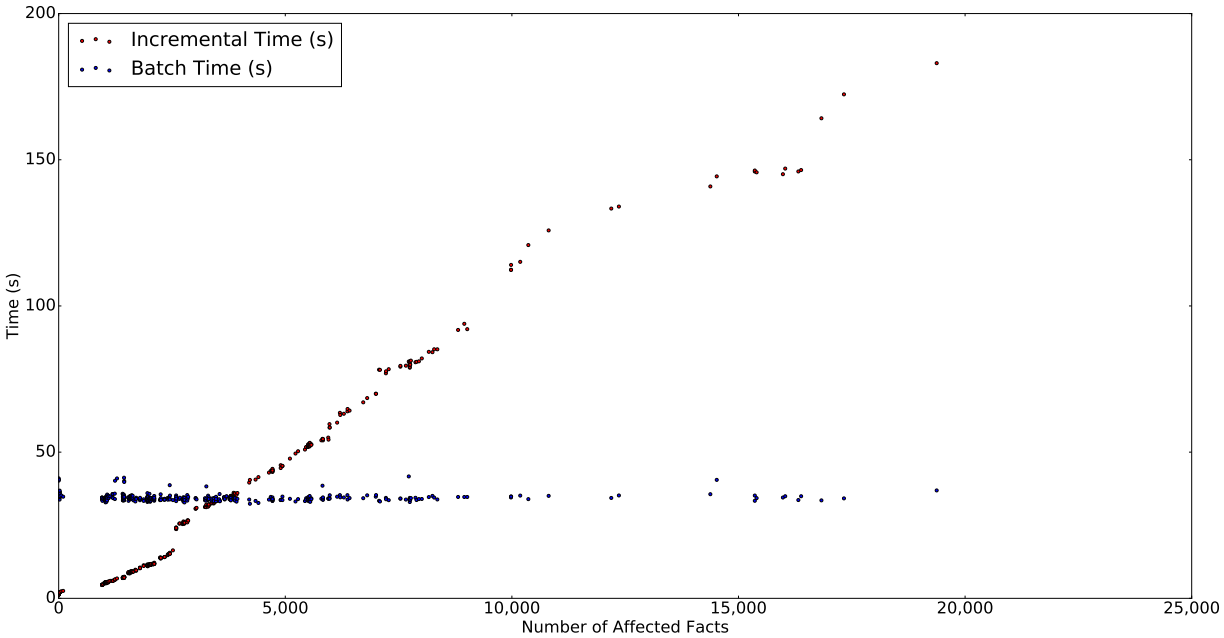


Figure 4.8: Incremental Analysis Growth for Query 4.2

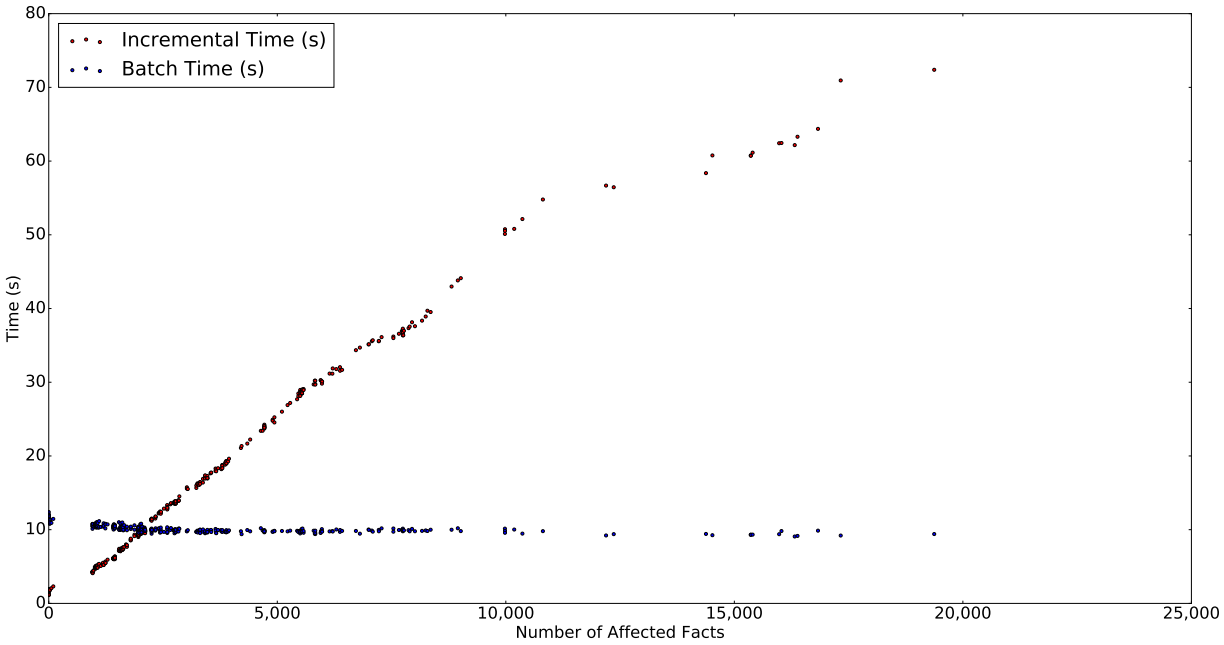


Figure 4.9: Incremental Analysis Growth for Query 4.3

Our next step was to investigate the source of this exponential growth. First, we wanted to see if the underlying explosions in computation time could be narrowed down to a file-level δ by measuring how δ's in Δ affect the number of derived facts to be added to the result set and the number of derived facts that could potentially be removed. These measurements are shown in Figures 4.10 and 4.12 and 4.14, where we can see sudden explosions in both the number of facts to be added and to be deleted when certain δ's are added to the subsets of Δ. To confirm that these δ's are the cause of the explosions, we measured how each individual $\delta \in \Delta$ affected the number of derived facts to be added to the result set and the number of derived facts that could potentially be removed. These measurements are shown in Figures 4.11, 4.13 and 4.15, where the right-skewed distributions in Figures 4.11a, 4.11b, 4.13a, 4.13b, 4.15a and 4.15b indicate that these δ's have a significantly larger impact on the number of derived facts to be added to the result set and the number of derived facts that could potentially be removed.

We tried to narrow down the cause of these explosions further to a tractable subset of facts. In particular, we studied the following explosions: the first jump in the number of additions made in Figure 4.12, the jump in the number of additions made from less than 400,000 to around 600,000 in Figure 4.10, the jump in the number of additions made from around 600,000 to around 1,000,000 in Figure 4.10, the jump in the number of initial removals from less than 200,000 to around 400,000 in Figure 4.10 and the first jump in the number of additions made in Figure 4.14.

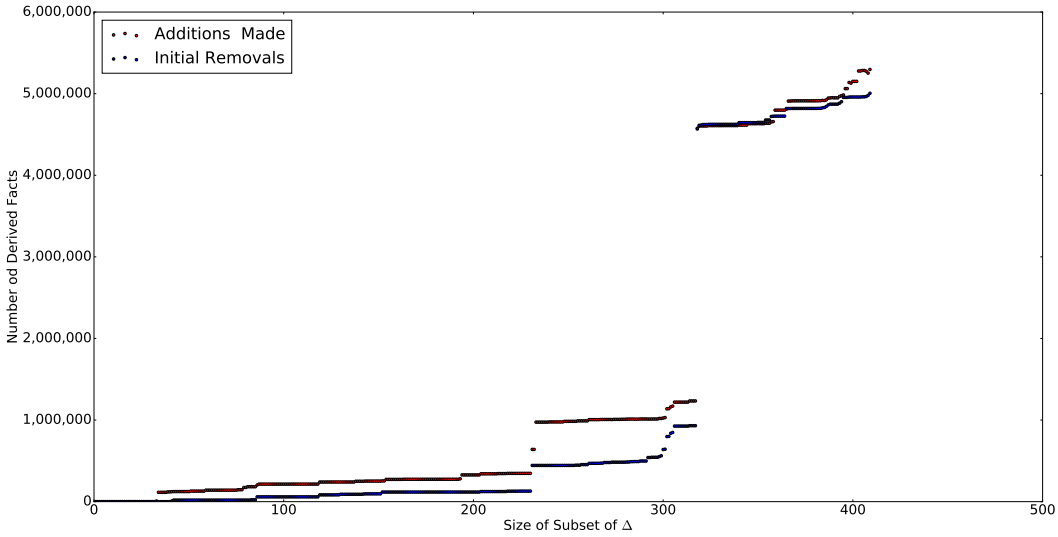
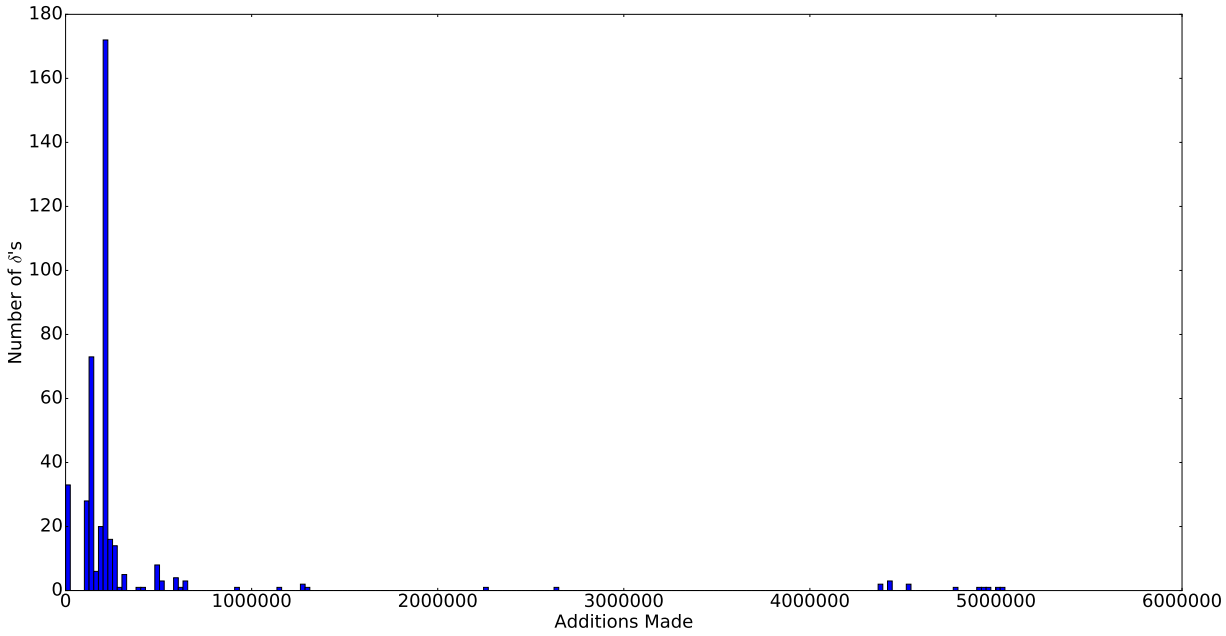
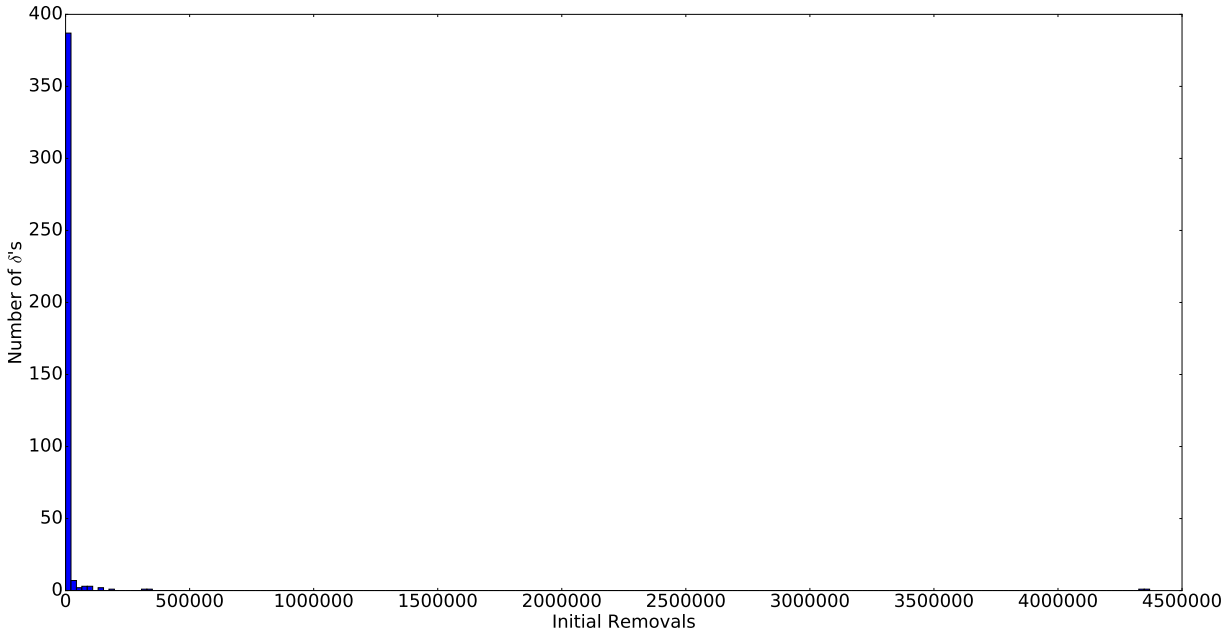


Figure 4.10: Explosions of Derived Changes to *Analysis(linux.fb)* as subsets of Δ increase (Query 4.1)



(a) Derived Additions to *Analysis(linux.fb)* per $\delta \in \Delta$ (Query 4.1)



(b) Derived Removals to *Analysis(linux.fb)* per $\delta \in \Delta$ (Query 4.1

Figure 4.11: Derived Changes to to *Analysis(linux.fb)* per $\delta \in \Delta$ (Query 4.1)

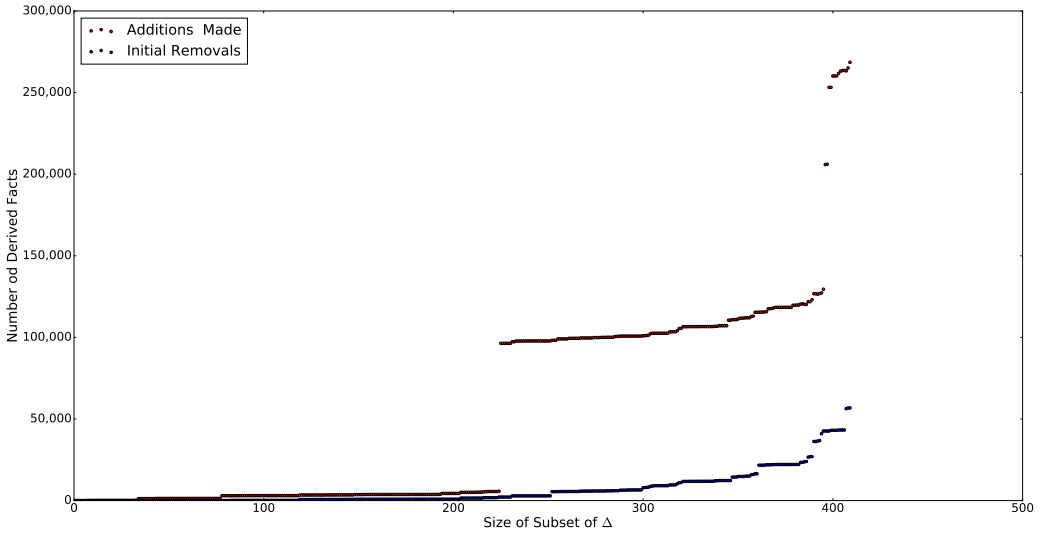
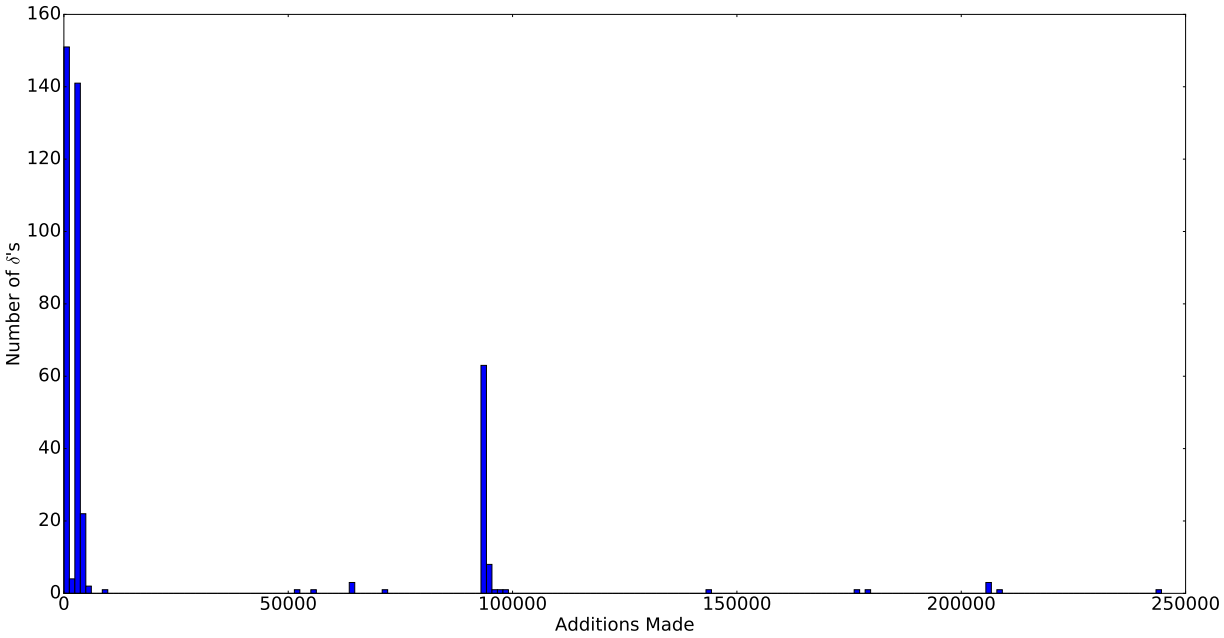
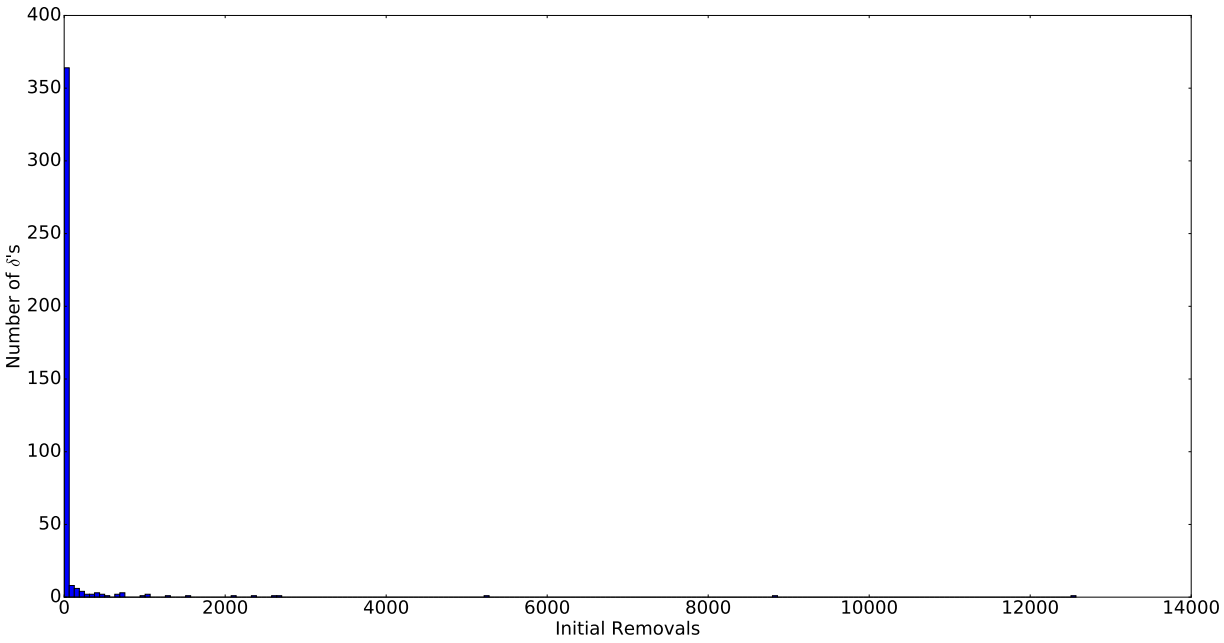


Figure 4.12: Explosions of Derived Changes to *Analysis(linux.fb)* as subsets of Δ increase (Query 4.2)



(a) Derived Additions to *Analysis(linux.fb)* per $\delta \in \Delta$ (Query 4.2)



(b) Derived Removals to *Analysis(linux.fb)* per $\delta \in \Delta$ (Query 4.2)

Figure 4.13: Derived Changes to to *Analysis(linux.fb)* per $\delta \in \Delta$ (Query 4.2)

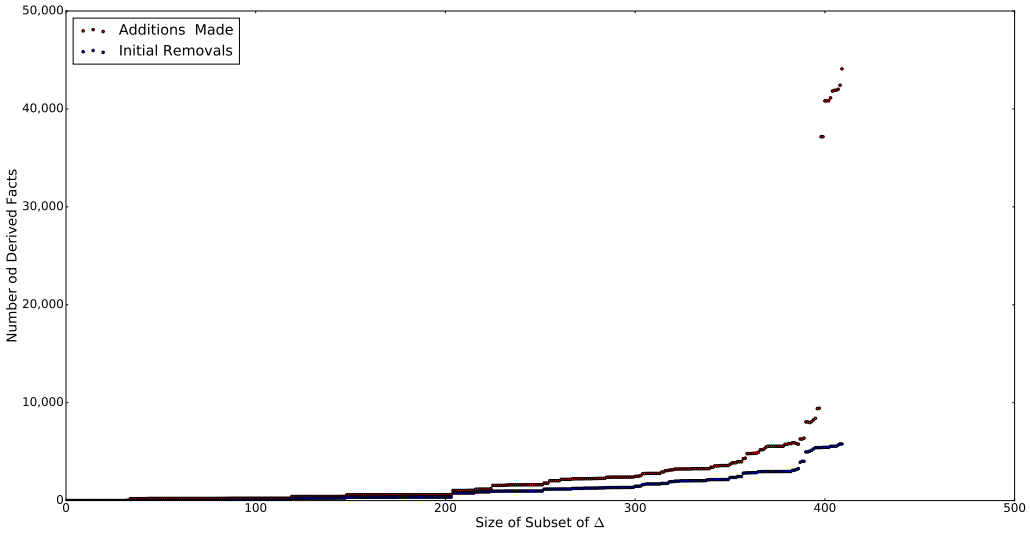
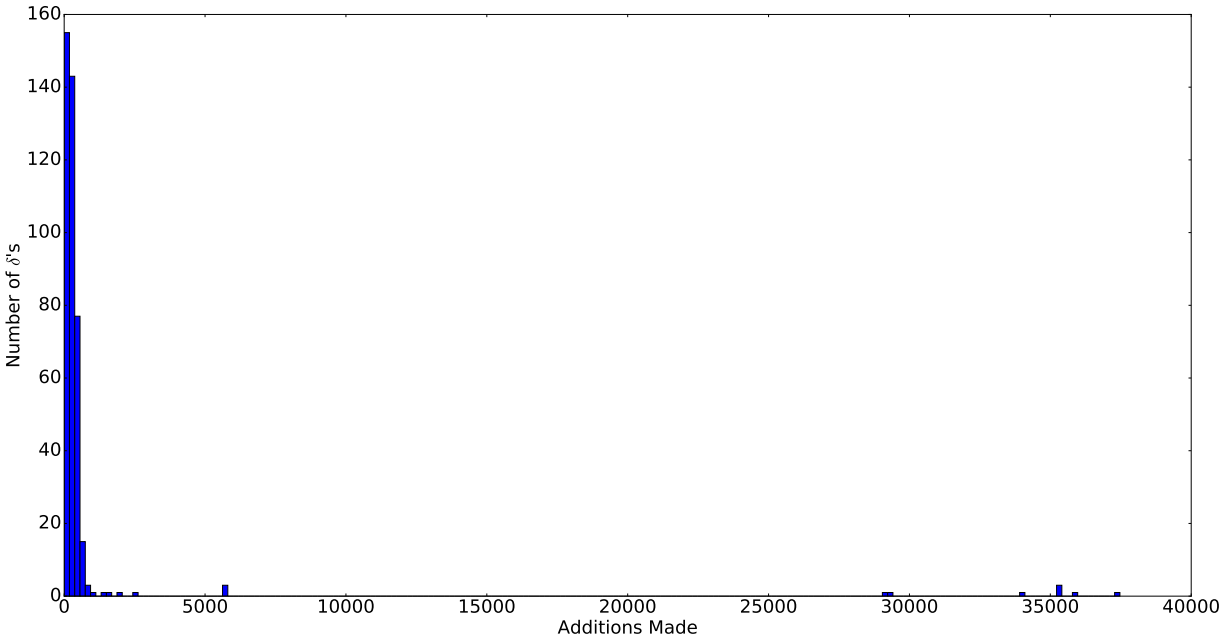
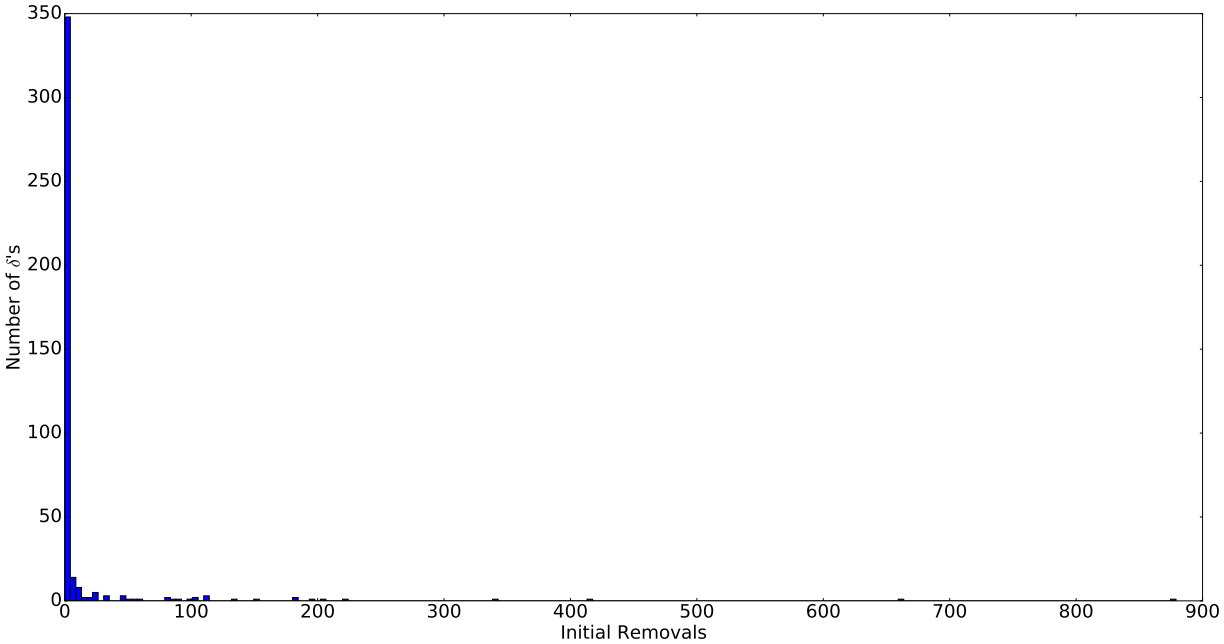


Figure 4.14: Explosions of Derived Changes to *Analysis(linux.fb)* as subsets of Δ increase (Query 4.3)



(a) Derived Additions to *Analysis(linux.fb)* per $\delta \in \Delta$ (Query 4.3)



(b) Derived Removals to *Analysis(linux.fb)* per $\delta \in \Delta$ (Query 4.3)

Figure 4.15: Derived Changes to to *Analysis(linux.fb)* per $\delta \in \Delta$ (Query 4.3)

For each δ that caused these jumps, we manually identified a subset of facts as the potential cause. Such facts were 'centroids' of an analysis; they had relatively higher fan-in and fan-out, resulting in numbers of derived facts being added or removed that were at least double and sometimes orders of magnitudes larger than 'non-centroids'. With the potential causes identified, we then re-ran an Incremental Analysis with these facts excluded to verify that our identified transitive explosions are reduced in their absence, which Figures 4.16, 4.17 and 4.18 illustrate. Thus, the exponential growth in an Incremental Analysis' computation time can be traced to individual facts.

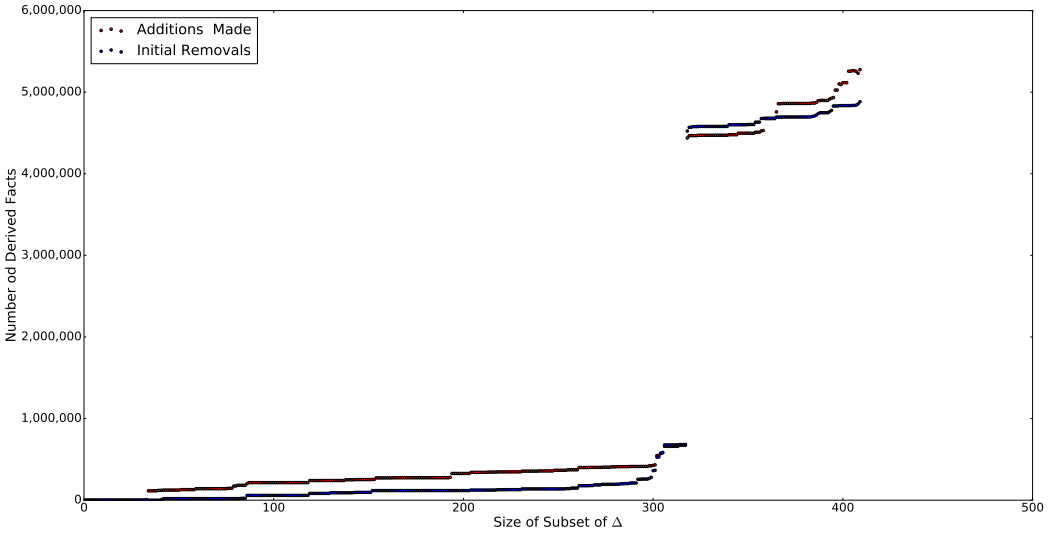


Figure 4.16: Reduced Explosions of Derived Changes to *Analysis(linux.fb)* as subsets of Δ increase (Query 4.1)

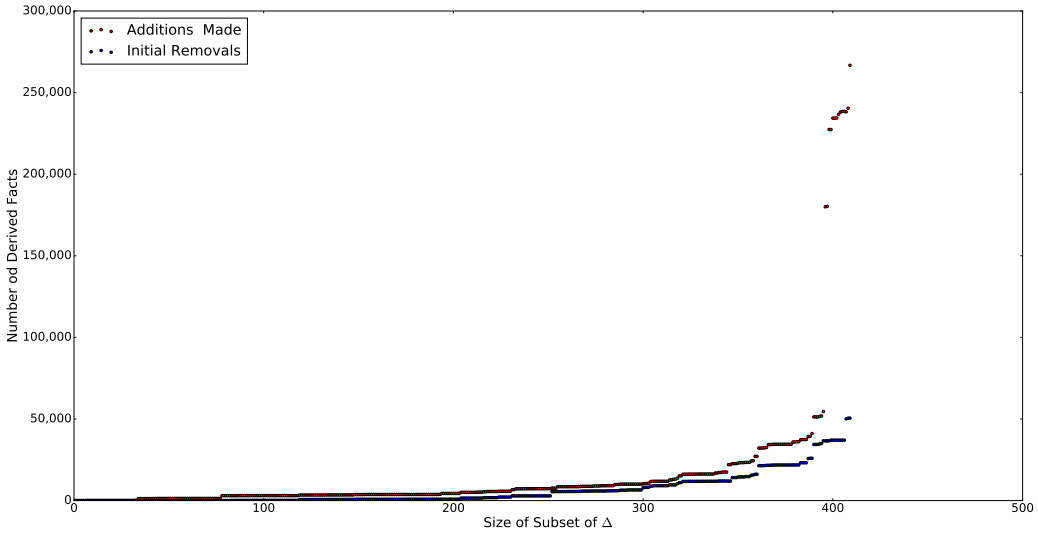


Figure 4.17: Reduced Explosions of Derived Changes to *Analysis(linux.fb)* as subsets of Δ increase (Query 4.2)

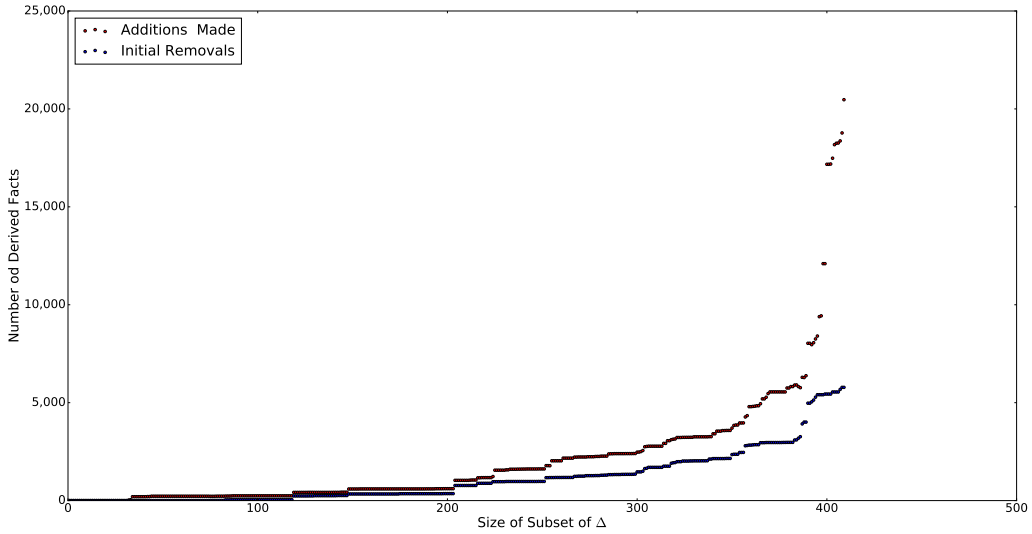


Figure 4.18: Reduced Explosions of Derived Changes to *Analysis(linux.fb)* as subsets of Δ increase (Query 4.3)

4.3 Threats to Validity

The major threat to validity for this study is that our results and conclusions are based on data generated from one software project. It may be possible that our findings are specific to the project corpus. Ideally, our results could be replicated on a variety of software projects in a variety of application domains and languages, lending support to this study's generalizability.

Another threat to validity concerns the manual analysis done in breaking down δ. Although we were able to identify facts that caused the transitive explosions we investigated, some were still left unexplored. As a matter of completeness, we could conduct a similar manual analysis on these jumps.

Finally, our comparison of the IAP and the NAP was performed using two Linux versions rather than a sequence of Linux versions. It may be the case that our findings are specific to these two particular versions. To mitigate this threat, a comparison between IAP and the NAP could be done with respect to the same series of Linux versions as the comparison between the IEP and NEP.

4.4 Summary

In this chapter, we discussed a case study that we performed using the Linux Kernel to compare the speed and accuracy of the IEP and IAP against the NEP and the NAP, respectively. To compare the IEP against the NEP, we applied both processes to update extracted factbases of 'allNoConfig' builds of the Linux Kernel from versions 4.0 to 5.5. To compare the IAP against the NAP, we applied both processes to update an existing set of analysis results using increasing subsets of Δ for three different analysis queries. Overall, we found the IEP to be 54-65% quicker than the NEP, the IEP scaling linearly with respect to the size of an input Δ. In contrast, the results of the IAP were mixed. While the IAP was quicker than the NAP for smaller $\Delta's$, it was highly sensitive to 'centroids' of facts that tended to expand an analysis' search space exponentially, resulting in exponential increases in computation time. Accordingly, we found that the IAP scaled linearly with respect to the number of affected facts in the best case, and followed an S-shaped growth in the worst case.

Chapter 5

Conclusions

Our work presents three major contributions:

- A framework for incrementally updating previously extracted System Factbases. This framework builds upon the Separate Fact Extraction pipeline to account for the following changes: modifications to source files, removal of source files and addition of source files.
- A framework for incrementally updating a set of analysis results. This framework is in many ways similar to a Rete network, allowing analyses to be updated only when their underlying base facts have been written to or removed. We introduce the notion of *Meta Facts* to represent such changes.
- A case study that evaluates the effectiveness and feasibility of our proposed frameworks. The case study was conducted on 'allNoConfig' builds of the Linux kernel, consisting of roughly 200,000 SLOC, by modifying an existing extraction & analysis pipeline to incorporate our incremental approaches. We compared the batch and incremental approaches to updating previously extracted factbases, and the batch and incremental approaches to updating a previous set of analysis results. Further, we investigated the sources of bottlenecks in our Incremental Analyses.

Overall, we found that our approach to incrementally updating a previously extracted factbase is 54%-65% quicker than a batch update, with both updates producing identical factbases. The results of our Incremental Analysis (IAP), on the other hand, was mixed. While IAP was quicker than a batch analysis for smaller Δ's, it was highly sensitive to

'centroids' of an analysis; facts with relatively higher fan-in and fan-out tended to expand the search space exponentially, resulting in exponential increases in computation time, with the effect of these 'centroids' being magnified by transitive closure. This means that IAP scales linearly with respect to the number of affected facts in the best case, and in the worst case experiences an S-shaped growth.

5.1 Future Work

Currently, the IAP is limited to relational operators, specifically union, composition, natural join, transitive closure and inversion. The logical next step is to add support for node/entity and property/attribute operators, such as filtering and selection. The most interesting area of future work, however, surrounds the 'centroids' in a δ. It may be worth studying them to understand how they relate to their underlying source-code entities. We could then leverage this information to predict when an Incremental Analysis ought to outperform a Batch Analysis. For example, it may be the case that our Incremental Analysis is better suited to certain types of software projects.

Another path we could take is to look into mitigating the effects of 'centroids' on Incremental Analyses. This could be done by creating summaries of these 'centroids' to prevent an Incremental Analysis from having to re-explore their search space. In theory, we could expand this concept to the entire factbase for a similar effect. Alternatively, a hybrid of Batch and Incremental Analysis using these summaries may reap the benefits of both, the Batch Analysis only being applied when detected changes involve a 'centroid'.

References

[1] Auckland threatens to eject Lime scooters after wheels lock at high speed — World news — The Guardian.

[2] Infusion Pump Software Safety Research at FDA — FDA.

[3] Linux Kernel At 19.5 Million Lines Of Code, Continues Rising - Phoronix.

[4] Linux kernel in 2011: 15 million total lines of code and Microsoft is a top contributor — Ars Technica.

[5] The history of Linux: how time has shaped the penguin: Page 2 — TechRadar.

[6] US aviation authority: Boeing 787 software bug could cause 'loss of control' — Technology — The Guardian.

[7] Renzo Angles. The Property Graph Database Model. Technical report, 2018.

[8] Andrew Begel and Nachiappan Nagappan. Usage and perceptions of agile software development in an industrial context: An exploratory study. Technical Report MSR-TR-2007-09, September 2007. First International Symposium on Empirical Software Engineering and Metrics.

[9] William C. Benton and Charles N. Fischer. Interactive, scalable, declarative program analysis: From prototype to implementation. In *PPDP'07: Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 13–24, New York, New York, USA, 2007. ACM Press.

[10] Thomas Beyhl and Holger Giese. *Efficient and scalable graph view maintenance for deductive graph databases based on generalized discrimination networks.* 08 2015.

[11] David Binkley and David. Source Code Analysis: A Road Map. In *Future of Software Engineering (FOSE '07)*, pages 104–119. IEEE, may 2007.

[12] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, pages 243–261, New York, New York, USA, 2009. ACM Press.

[13] Yih Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334, 1990.

[14] Michael L. Collard, Michael J. Decker, and Jonathan I. Maletic. Lightweight Transformation and Fact Extraction with the srcML Toolkit. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, pages 173–184. IEEE, sep 2011.

[15] Patrick Cousot and Radhia Cousot. Modular Static Program Analysis. pages 159–179. Springer, Berlin, Heidelberg, 2002.

[16] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings - International Conference on Software Engineering*, pages 391–400, 2008.

[17] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, sep 1982.

[18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1433–1445. Association for Computing Machinery, may 2018.

[19] Gábor Szárnyas. *Query, Analysis, and Benchmarking Techniques for Evolving Property Graphs of Software Systems*. PhD thesis.

[20] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, feb 2005.

[21] Oshini Goonetilleke, David Meibusch, and Ben Barham. Graph data management of evolving dependency graphs for multi-versioned codebases. In *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017*, pages 574–583. Institute of Electrical and Electronics Engineers Inc., nov 2017.

[22] Sergio Greco and Cristian Molinaro. Datalog and Logic Databases. *Synthesis Lectures on Data Management*, 7(2):1–169, nov 2015.

[23] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data - SIGMOD '95*, pages 328–339, New York, New York, USA, 1995. Association for Computing Machinery (ACM).

[24] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. *ACM SIGMOD Record*, 22(2):157–166, jan 1993.

[25] Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. CodeQuest: Scalable source code queries with datalog. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4067 LNCS, pages 2–27. Springer Verlag, 2006.

[26] Ric Holt. TA: The Tuple Attribute Language, 1997.

[27] Richard C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *Reverse Engineering - Working Conference Proceedings*, pages 210–219. IEEE Comp Soc, 1998.

[28] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings - International Conference on Software Engineering*, pages 672–681, 2013.

[29] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '05, page 1–12, New York, NY, USA, 2005. Association for Computing Machinery.

[30] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *The Journal of Systems and Software*, 1(C):213–221, jan 1979.

[31] Yuan Lin. *Completeness of Fact Extractors and a New Approach to Extraction with Emphasis on the Refers-to Relation*. PhD thesis, aug 2008.

[32] Yuan Lin and Richard C. Holt. Formalizing Fact Extraction. *Electronic Notes in Theoretical Computer Science*, 94:93–102, may 2004.

[33] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Maintenance of datalog materialisations revisited. *Artificial Intelligence*, 269:76 – 136, 2019.

[34] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 20, pages 18–27. ACM, oct 1995.

[35] Bryan J. Muscedere, Robert Hackman, Davood Anbarnam, Joanne M. Atlee, Ian J. Davis, and Michael W. Godfrey. Detecting Feature-Interaction Symptoms in Automotive Software using Lightweight Analysis. In *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*, pages 175–185. Institute of Electrical and Electronics Engineers Inc., mar 2019.

[36] Christian R. Prause, René Reiners, and Silviya Dencheva. Empirical study of tool support in highly distributed research projects. In *Proceedings - 5th International Conference on Global Software Engineering, ICGSE 2010*, pages 23–32. IEEE Computer Society, 2010.

[37] H. Prähofer, F. Angerer, R. Ramler, and F. Grillenberger. Static code analysis of iec 61131-3 programs: Comprehensive tool support and experiences from large-scale industrial application. *IEEE Transactions on Industrial Informatics*, 13(1):37–47, Feb 2017.

[38] Xiaolei Qian and Gio Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[39] Rudolf Ramler, Georg Buchgeher, Claus Klammer, Michael Pfeiffer, Christian Salomon, Hannes Thaller, and Lukas Linsbauer. Benefits and drawbacks of representing and analyzing source code and software engineering artifacts with graph databases. In Dietmar Winkler, Stefan Biffl, and Johannes Bergsmann, editors, *Software Quality: The Complexity and Challenges of Software Engineering and Software Quality in the Cloud*, pages 125–148, Cham, 2019. Springer International Publishing.

[40] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470 – 495, 2009.

[41] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from Building Static Analysis Tools at Google, 2018.

[42] Robert W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings - International Conference on Software Engineering*, pages 83–92. Publ by IEEE, oct 1991.

[43] Ramy Shahin, Marsha Chechik, and Rick Salay. Lifting datalog-based analyses to software product lines. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 39–49. ACM, 2019.

[44] Dániel Stein and István Ráth. Incremental Static Analysis of Large Source Code Repositories Scientific Students' Association Report. Technical report, 2014.

[45] Hannes Thaller. Probabilistic software modeling. *CoRR*, abs/1806.08942, 2018.

[46] Alexandra Witze. Software error doomed Japanese Hitomi spacecraft, apr 2016.

[47] Jingwei Wu. *Open Source Software Evolution and Its Dynamics*. PhD thesis, 2006.

[48] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 590–604. Institute of Electrical and Electronics Engineers Inc., nov 2014.

APPENDICES

Appendix A

Algorithms used in IEP

A.1 Diff & Replace

Algorithm 1 below is used in the IEP's *Diff and Replace* process, where:

> fb_{old} and fb_{new} are strings that represent a factbase's filename
>
> $fbStream_{old}$ and $fbStream_{new}$ are input streams
>
> generateDiff(fb_{old}, fb_{new}) — Compares the contents of the factbases with filenames fb_{old} and fb_{new} using a merge-like routine, producing a file-level δ containing facts that are in fb_{new} but not in fb_{old}, prefixed with a '+', facts that are in fb_{old} but not in fb_{new}, prefixed with a '-'. Facts that are in both fb_{old} and fb_{new} are compared at the attribute level, with the differences being added to δ.
>
> rename(oldName, newName) renames the file *oldName* to *newName*
>
> remove(filename) removes a file with name *filename*
>
> inputStream(filename) creates an input stream to the file with name *filename*
>
> nextLine(istream) reads the next line in an input stream *istream*

:

Algorithm 1 Diff and Replace

```
1: procedure DIFF_AND_REPLACE(fbList_old, fbList_new, Δ)
2:     fbStream_old ← inputStream(fbList_old)
3:     fbStream_new ← inputStream(fbList_new)
4:     fb_old ← nextLine(fbStream_old)
5:     fb_new ← nextLine(fbStream_new)
6:     Δ ← ∅
7:     while !fbStream_old.eof() and !fbStream_new.eof() do
8:         if fb_old < fb_new then
9:             δ ← generateDiff(fb_old, ∅)
10:            Δ ← Δ ∪ δ
11:            remove(fb_old)
12:            fb_old ← nextLine(fbStream_old)
13:        else if fb_old > fb_new then
14:            δ ← generateDiff(∅, fb_new + ".new")
15:            Δ ← Δ ∪ δ
16:            rename(fb_new + ".new", fb_new)
17:            fb_new ← nextLine(fbStream_new)
18:        else if exists(fb_new + ".new") then
19:            δ ← generateDiff(fb_old, fb_new + ".new")
20:            Δ ← Δ ∪ δ
21:            remove(fb_old)
22:            rename(fb_new + ".new", fb_new)
23:            fb_new ← nextLine(fbStream_new)
24:            fb_old ← nextLine(fbStream_old)
25:        end if
26:    end while
```

```
27:     while !fbStream_old.eof() do
28:         δ ← generateDiff(fb_old, ∅)
29:         Δ ← Δ ∪ δ
30:         remove(fb_old)
31:         fb_old ← nextLine(fbStream_old)
32:     end while
33:     while !fbStream_new.eof() do
34:         δ ← generateDiff(∅, fb_new + ".new")
35:         Δ ← Δ ∪ δ
36:         rename(fb_new + ".new", fb_new)
37:         fb_new ← nextLine(fbStream_new)
38:     end while
39:     remove(fbList_old)
40:     rename(fbList_new, fbList_old)
41: end procedure
```

A.2 Update

The following algorithms are used in the *Update* step of the IEP.

Applying Removals

Algorithm 2 Applying Removals

```
1: procedure APPLY_REMOVALS(Δ, V, E)
2:     for all δ ∈ Δ do
3:         for all (edge, property) ∈ δ['edgePropertyRemoved'] do
4:             if edge ∈ E then
5:                 removeProperty(edge, property)
6:             end if
7:         end for
8:         for all edge ∈ δ['edgeRemoved'] do
9:             if edge ∈ E and edge['instances'] = 0 then
10:                E ← E \ {edge}
11:            end if
12:        end for
13:        for all (node, property) ∈ δ['nodePropertyRemoved'] do
14:            if node ∈ V then
15:                removeProperty(node, property)
16:            end if
17:        end for
18:        for all node ∈ δ['nodeRemoved'] do
19:            if node ∈ V and node['instances'] = 0 then
20:                E ← E \ edges(node)
21:                V ← V \ {node}
22:            end if
23:        end for
24:    end for
25: end procedure
```

Applying Additions and Mods

Algorithm 3 Applying Additions And Mods

```
1: procedure APPLY_ADD_AND_MOD(Δ, V, E)
2:     for all δ ∈ Δ do
3:         V ← V ∪ δ['nodeAdded']
4:         for all (node, property) ∈ δ['nodePropertyAdded'] do
5:             if node ∈ V then
6:                 addProperty(node, property)
7:             end if
8:         end for
9:         for all (node, property) ∈ δ['nodePropertyMod'] do
10:            if node ∈ V then
11:                modifyProperty(node, property)
12:            end if
13:        end for
14:        for all edge ∈ δ['edgeAdded'] do
15:            if edge.src ∉ V then
16:                V ← V ∪ edge.src_virtual
17:            end if
18:            if edge.dst ∉ V then
19:                V ← V ∪ edge.dst_virtual
20:            end if
21:            E ← E ∪ edge
22:        end for
23:        for all (edge, property) ∈ δ['edgePropertyAdded'] do
24:            if edge ∈ E then
25:                addProperty(edge, property)
26:            end if
27:        end for
28:        for all (edge, property) ∈ δ['edgePropertyMod'] do
29:            if edge ∈ E then
30:                modifyProperty(edge, property)
31:            end if
32:        end for
33:    end for
34:    removeVirtualNodes(V, E)
35: end procedure
```