Provenance Tracking for Static Analysis with Datalog

Sarah Sallinger

School of Computer and Communication Sciences

A thesis submitted for the degree of Master of Science at
École polytechnique fédérale de Lausanne

31 July 2019

Supervisor
Prof. Viktor Kunčak
EPFL / LARA

Company Supervisor
Marcelo Sousa, DPhil.
SonarSource

Acknowledgements

First of all, a huge thanks to Marcelo Sousa, my mentor throughout my internship at SonarSource and advisor for this project, and Viktor Kunčak, who has been my advisor throughout my studies at EPFL. Thank you for supporting me through all the ups and downs and for all your patience and helpfulness. It was a fun semester working on a great topic, and I have learnt a lot.

Furthermore, I would like to sincerely thank Bernhard Scholz and David Zhao from the University of Sydney for providing valuable insights into the inner workings of Soufflé and for giving me valuable feedback on my work.

A special thanks to Katharina and Lydia for helping me proofread this thesis, I hope it was an interesting read!

Finally, a big thank you to the other members of SonarSource and of the LARA lab at EPFL, to my family, and to my friends for your invaluable support throughout this semester. Thank you for being great companions and for keeping up the good vibes!

Abstract

Logic programming languages such as Datalog are gaining popularity for industrial static program analysis. This rise in popularity is due to the ease of expressing analyses in a declarative manner and to the availability of Datalog solvers that allow for performance characteristics similar to those of hand crafted analyses.

A major challenge in using Datalog for program analysis is the generation of valuable information about generated alarms to give useful feedback to the users. A first step towards obtaining this information is the computation of provenance information for given analysis alarms. The state-of-the-art Datalog engine Soufflé provides this functionality in a system component that allows users to construct proof trees for arbitrary alarms.

Other than the Datalog evaluation itself, this component is not fine-tuned for performance which results in unnecessarily long proof tree construction times. Soufflé's proof tree construction mechanism relies on annotations that are added to the generated tuples during Datalog evaluation, describing the height of the tuple's proof tree.

This thesis introduces *subtree-heights provenance*, an alternative proof tree construction mechanism that additionally annotates tuples with the heights of the proof trees of the tuples that where used in the generation, i.e. the heights of the first level subtrees.

The alternative provenance mechanism was implemented in Soufflé and evaluated on a set of different Datalog program analyses over real-world programs, showing significant reductions of the proof tree computation times in all setups and reaching reductions of up to 80% in some setups.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

Logic programming languages are gaining more and more popularity as domain specific languages for large-scale applications including static program analysis [18, 8, 4]. Logic programs have declarative semantics, that is, the program describes the intended results rather than specifying detailed computational steps of how the result is computed. For developers of static analyses, this is advantageous as analyses can be expressed very succinctly, simplifying the development process and reducing development time.

A particular logic programming language that plays an important role in this context is Datalog [12]. Datalog is a database query language for deductive databases, i.e. databases that use rules to derive new facts from given input facts. The advantage of Datalog over conventional database languages based on relational algebra is its natural support for recursive queries [17]. Syntactically, Datalog is a subset of Prolog and while its expressive power is strictly smaller, it can be evaluated by efficient bottom up algorithms [1].

While Datalog have been proposed for program analysis by the research community several years ago [30, 8], the interest in the topic is currently revived due to advances in Datalog solvers as they are able to compete with the performance of hand-crafted analysers [18]. This rise in interest in Datalog-based analyses is noticeable not only in the research community, but also in the industry where notable examples include the code analysis provider Semmle [4] and static analysers developed at Oracle [27].

The project of this thesis was conducted as part of an internship at SonarSource, a company that develops products for continuous code quality for more than 20 different programming languages. Given that the analysers included in the products are mostly written in general purpose programming languages such as Java, they become complex pieces of software which are expensive to develop and maintain. The potential of using Datalog as an alternative language for developing analysers in this context was the main motivation for this project.

While the advantages of expressing analyses in a declarative way seemed apparent, the goal of the project was to look into some concerns that arise when using Datalog for industrial

analysers. The first concern was whether the analysers can actually meet the scalability requirements for analysing large codebases with millions of lines of code.

In order to best meet the scalability requirements, we decided to focus our exploration on the open source Datalog engine Soufflé [18] which specifically targets static analyses. It synthesizes C++ code from Datalog specifications by doing specialization steps based on Futamura projections [15]. The generated C++ code implements a specialized semi-naïve algorithm [1] for bottom up Datalog evaluation. The code is targeted at parallel execution on shared memory multi-core machines.

As reported in [18], the static analysers synthesised by Soufflé achieve a performance similar to hand written-analyzers. Soufflé is successfully used in several large scale projects, e.g. security analysis [27], smart contract analysis [9], and in the pointer and taint analysis framework Doop [2]. This large body of applications around Soufflé served as indication that a satisfiable scalability can be achieved.

The second big concern that arose in the exploration is whether analyses written in Soufflé provide sufficient information about generated alarms in order to be able to give useful feedback to users. We started our investigation by looking into the output of a simple taint analysis written in Soufflé. Taint analyses detect whether values can flow from sources, i.e. locations reading unfiltered user input, to sinks, i.e. locations in the code executing sensitive commands. A core feature of the taint analyser developed at SonarSource is that, in case a taint flow from a source to a sink is located, a data flow path that the tainted value takes through the code is displayed to the user. In contrast to this, evaluating the rules of the taint analyser produced by Soufflé only yields pairs of reachable sources and sinks without providing further information about the path between the two.

In Soufflé, a first step towards giving additional information about analysis alarms, i.e. outputs of the Datalog evaluation, is to compute provenance information for the output tuples of the analysis. Provenance information provides insights about why specific tuples were added to the output of a Datalog program. Soufflé provides functionality to compute provenance in the form of proof trees [33]. A tuple's proof tree explains how the tuple was derived from the input facts by rule applications.

Our first experiments showed that Soufflé's proof tree generator worked fine for small examples, but was tremendously slow for computing proof trees of more complex analyses. For example, one particular proof tree computation did not complete under one hour for an output tuple of a taint analysis included in Doop [2] on a small Java program with ten lines of code. After discussing those performance issues with the developers of Soufflé, it turned out that they were caused by a bug in the version of Soufflé that was used in the experiments. However, the first experiments motivated us to look into the algorithms of Soufflé's provenance component and into possible optimizations that seemed to have potential for improving the performance of proof tree construction even after the bug causing the initial performance issues was fixed.

In this thesis we introduce *subtree-heights provenance*, an alternative way of computing proof trees for outputs of Datalog analyses. Soufflé's proof tree computation relies on a two step approach: first, it adds proof annotations during bottom up evaluation and then uses these annotations in a top down proof tree construction algorithm. In particular, every generated tuple is annotated with the identifier of the rule that was used for its construction and the height of its proof tree [33]. The key change of our approach is to additionally store the heights of the proof trees of the tuples that where used in the generation, i.e. the heights of the first level subtrees.

The main research hypothesis in this thesis is that annotating tuples with the heights of the first level subtrees in Soufflé dramatically improves the runtime of proof tree construction.

1.1 Contributions

This thesis makes the following contributions:

- We introduce *subtree-heights provenance* which is a more efficient provenance computation algorithm than Soufflé's current strategy storing additional annotations at evaluation time.
- We establish the correctness of the new evaluation strategy through a new proof tree metric and a proof that this metric satisfies the requirements of Soufflé's provenance framework.
- We report on the implementation of the new evaluation strategy and the new proof tree construction algorithm in Soufflé. This includes the required modifications to the phase which adds proof annotations but also to big parts of the other phases of Soufflé's synthesiser as the proof annotations are integrated tightly into the data structure for storing generated tuples and many parts of the code are built on the assumption that there are exactly two proof annotations. The implementation is publicly accessible at https://github.com/ssallinger/souffle.
- We present an experimental evaluation on the impact of the new provenance computation strategy on time and memory usage at evaluation as well as at proof tree construction time. For the experiments, we implemented two variations of reachability analysis over control flow graphs (CFGs). We built a translation unit that translates an internal representation of CFGs used by SonarSource analysers to Datalog, giving access to a front-end for Java, PHP and C# programs. For the experiments, the analyses were run over a set of mature real world open source projects in Java and PHP. For both variations of the analysis, the new provenance computation yielded a significant speed up in the proof tree computation time, reaching a speed up as high as 80% for one of the two variants. The overhead in evaluation time was between 20% and 50% depending on the considered setup. The memory overhead ranged from 50% to 180%.

```
1  //Input: edge relation
2  .decl edge(x:number, y:number)
3
4  edge(1, 2).
5  edge(2, 3).
6  edge(3, 4).
7  edge(4, 5).
8  edge(5, 4).
9
10 //Output: path relation
11 .decl path(x:number, y:number)
12 .output path
13
14 //r1:
15 path(x, y) :-
16     edge(x, y).
17
18 //r2:
19 path(x, y) :-
20     path(x, z),
21     edge(z, y).
```

Listing 1.1 – Example datalog program

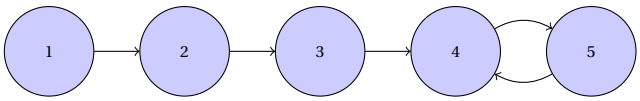

Figure 1.1 – Input grph for transitive closure example

1.2 Motivation and Running Example

This section presents the key idea of our provenance computation on a simple example. We use this example throughout the thesis to illustrate concepts introduced in this work.

Datalog programs specify logical rules over a set of relations. Consider the Datalog program shown in Listing 1.1. It specifies rules for deriving the `path` relation, a relation representing the transitive closure of a graph, from the `edge` relation which contains the input facts describing the edges of the graph. In this small example, the facts are directly encoded in the program. Alternatively, Soufflé provides an option for reading input facts from external files. The graph used in this example is displayed in Figure 1.1.

The `path` relation represents the set of pairs of nodes in the graph that are connected by a path. In this example, two rules are used to add tuples to this relation. The first rule, rule `r1`,

```
1 edge(1, 2)
2 -----------(r1)
3 path(1, 2)       edge(2, 3)
4 --------------------------(r2)
5 path(1, 3)                          edge(3, 4)
6 ----------------------------------------(r2)
7 path(1, 4)                                        edge(4, 5)
8 ------------------------------------------------------------ (r2)
9 path(1, 5)
```

Listing 1.2 – Example proof tree

specifies that two nodes are connected by a path if there is an edge between them. The second rule, rule `r2` encodes the transitive property, i.e. there is a path between two nodes if there is a path from the first node to an intermediary node which is connected to the second node by an edge.

The `path` relation is marked as output relation which means that its tuples will be written to a file after completion of the evaluation. In general, only a subset of the computed relations is marked as output. The relations marked as output are what we consider the results of a Datalog program.

One of the tuples in the `path` relation is `path(1, 5)`. The tuple's proof tree is displayed in Listing 1.2. The nodes of the proof tree are labelled by tuples. Every level of the proof tree explains which rule was applied to which body tuples in order to generate the new tuples. For example, the root of the proof tree in Listing 1.2, denoted at the very bottom is tuple `path(1, 5)`. The next level explains that this tuple can be generated by applying rule `r2` to body tuples `path(1, 4)` and `edge(4, 5)`.

The crucial operation in proof tree construction is the search for the body tuples that were used to generate a given tuple. Soufflé annotates tuples with the height of their proof trees and the used rule during evaluation in order to guide this search. The height parameter is used to restrict the search for body tuples to tuples with smaller height than the current height. For example, the height of the proof tree of `path(1, 5)` is 4. In the search for body tuples, the algorithm looks for path tuples starting in 1 with heights smaller than 4. Hence, it considers tuples `path(1, 3)` and `path(1, 4)`.

The new provenance component introduced here additionally annotates tuples with the heights of their body tuples. In the considered example, this means that when the search for body tuples of `path(1, 5)` is started, it is known that the path body tuple has height 3. Hence, only tuple `path(1, 4)` is taken into account and the additional annotation is used to prune the search space. For large examples, this optimisation can dramatically reduce the runtime of the search. The search algorithms are explained in detail in Chapters 3 and 4.

1.3 Overview

This thesis is organised as follows: Chapter 2 surveys related work. Chapter 3 provides background information about Datalog evaluation and proof tree construction algorithms with a focus on the algorithms used in Soufflé. Chapter 4 presents the framework for the newly introduced provenance algorithms and their implementation in Soufflé. Chapter 5 presents experiments that demonstrate the effectiveness of the new provenance component and discusses the results. Chapter 6 concludes with an analysis of the main results and outlines ideas for future work.

2 Related Work

In this section, we describe related work about provenance in database systems and in particular about provenance for Datalog programs.

In the database research community, provenance is described as an explanation of the origin of data in a system [10, 13]. It provides a connection between output data and the input data that was used to generate it. In big database systems, provenance information can be used to establish the integrity of data [13] and for debugging purposes [20]. Two example systems that implement provenance for relational databases are *Trio* [7] and *Perm* [16].

Regarding the semantics of provenance, we can find several definitions in the literature. In [13], three different approaches are presented. The first approach is called *Why*-provenance which defines the provenance of a tuple as the set of input tuples that were involved in the tuple's generation. The second approach is referred to as *How*-provenance. It extends *Why*-provenance with the structure of a proof of how the input tuples were used to produce the output. This is formalized by associating each tuple with a special algebraic structure referred to as provenance semiring. The third approach is *Where*-provenance. In this approach, the origin of every element of a tuple is explained.

A natural way of describing provenance for Datalog programs is by specifying proof trees [3]. A tuple's proof tree fully explains how the tuple was derived from the input facts by rule applications. Hence, proof trees can be seen as a generalization of *How*-provenance.

Provenance in Datalog has been an active field of research [3, 11, 20, 22] outside the Soufflé engine. In [3], the idea of representing Datalog provenance in the form of proof trees is established. In [11], a debugging strategy for Datalog based on the principles of algorithmic debugging is presented. In [20], an alternative way of computing provenance information by generating a provenance-enriched rewriting is introduced. In [22], an approach for computing provenance for Datalog programs with negations is presented.

What distinguishes Soufflé's provenance component from these previous approaches is that, in general, those approaches rely on storing the full provenance information at evaluation

time which results in a memory overhead that is intractable for large-scale databases. It has been proposed to mitigate this high memory overhead by selectively storing provenance information for a predefined provenance query [14]. However, this means that the Datalog evaluation has to be repeated for every query of interest.

While the core application of most Datalog provenance systems is debugging, Datalog provenance is also used in different applications. Another line of work that relies on provenance information are Datalog based user-guided program analyses, which rank analysis alarms based on user feedback [23, 31, 24]. The optimized provenance computation presented in this thesis potentially is of special relevance to this line of work, as, in this setup, provenance information needs to be computed for all outputs of the program analysis. Hence, this computation takes a significant part of the overall analysis run time.

A presentation of Soufflé's provenance component is given in the following chapter. For a more in-depth discussion of how Soufflé's provenance compares to related work see [32, 33].

3 Preliminaries

3.1 Datalog evaluation

The two main Datalog evaluation strategies are bottom up and top down evaluation [1]. In bottom up evaluation, all possible facts are derived by applying the rules starting from the input facts. Top down evaluation proofs given facts of interest by applying the rules starting at those facts until the input facts are reached. In order to deal efficiently with large scale data, modern state of the art engines are mostly based on bottom up evaluation. As this is also the case for Soufflé, we give a short description here.

Standard bottom up evaluation of a Datalog program works by applying its constituent rules on a set of input tuples generating potentially new tuples. The new tuples are added to the current set and the rules are applied until a fixpoint is reached. For the example of computing paths in graphs introduced in Chapter 1, Listing 3.1 presents the iterations of the fixed point computation.

Formally, bottom up evaluation can be defined as a computation over a subset lattice defined over sets of tuples. Those sets of tuples are referred to as instances I. The naïve evaluation algorithm applies the so called immediate consequence operator on the current instance until fixpoint. The immediate consequence operator takes an instance and adds all tuples that can be generated by performing all possible rule applications on the tuples in the instance [1].

There are multiple variations of this basic evaluation strategy. In particular, Soufflé uses a more efficient strategy referred to as semi-naive evaluation [27]. The core advantage of semi-naive evaluation is that it avoids recomputing all tuples in every iteration.

3.1.1 Magic Set Transformation

Top down Datalog evaluation has an intrinsic performance advantage compared to bottom up evaluation if only a small fraction of the generated tuples is of interest for the user. For example in Soufflé, the user is only interested in the tuples in relations marked as output.

```
Iteration 1:
--------------
path(1, 2)
path(2, 3)
path(3, 4)
path(4, 5)
path(5, 4)

Iteration 2:
--------------
path(1, 3)
path(2, 4)
path(3, 5)
path(4, 4)
path(5, 5)

Iteration 3:
--------------
path(1, 4)
path(2, 5)

Iteration 4:
--------------
path(1, 5)
```

Listing 3.1 – Iterations of bottom up evaluation

```
1 edge(1, 2)
2 -------(r1)
3 path(1, 2) edge(2, 3)
4 --------------------(r2)
5 path(1, 3)
```

Listing 3.2 – Example proof tree

With standard bottom up evaluation all other relations will be computed, even if only a small fraction of their tuples might be involved in the generation of the output tuples.

The key idea of the magic set optimization is to rewrite the Datalog rules prior to bottom up evaluation, in order to avoid generating tuples that will never be needed for the generation of the output tuples [6]. The transformation specialises the rules based on constraints that appear in their bodies and in the bodies of their dependencies. The magic set transformation implemented in Soufflé is based on the algorithm presented in [5].

3.2 Provenance in Soufflé

In this section, we give an overview of Soufflé's current provenance component [33], regarding its functionality, its core algorithms and the implementation.

3.2.1 Proof Trees

Soufflé provides an option for computing provenance information in the form of minimal height proof trees. If Soufflé is run with the provenance option turned on, after the evaluation of the Datalog program, an interactive query interface is started. In the query interface, users can prompt the system to compute proof trees for any of the output tuples.

A proof tree provides a detailed explanation of which rules where involved in the generation of a tuple and which body tuples where used in the application of those rules. A proof tree for a tuple t is a tree where each node is labelled with a tuple. The root of the tree is labelled with t. The leaves of the tree are labelled with input facts. Every inner node labelled with a tuple t_0 is associated with a rule r such that r can be used to derive $t_0 :- t_1, ..., t_n$ and the direct child nodes of t_0 are labelled with $t_1, ..., t_n$.

For an example of a proof tree, see Listing 3.2. The tree describes why tuple `(1, 3)` was added to the `path` relation in the example Datalog program for computing transitive closures in graphs introduced in Section 1.2. It was generated by applying rule `r2` on body tuples `path(1, 2)` and `edge(2, 3)`. Tuple `path(1, 2)`, in turn, was generated applying rule `r1` on the input fact `edge(1, 2)`.

As standard bottom up evaluation does not yield any information about tuples's proof trees, Soufflé's computes proof trees in a two step approach. First, it adds proof annotations to generated tuples during bottom up evaluation and then uses these annotations in a top down proof tree construction algorithm. As the top down proof tree search is performed on an already populated Datalog instance, a subproof is guaranteed to exist for every tuple. This brings two core advantages compared to an approach relying excursively on top down evaluation. Firstly, the search algorithm does not have to use backtracking and can hence perform a faster search. Secondly, this allows for the construction of partial proof trees which turns out to be very useful for providing an interactive user interface.

The key operation in the second step of Soufflé's approach, proof tree construction, is to find for a given tuple which rule and body tuples were used in its generation. Unsurprisingly, there are various approaches to compute this information. One simple option is to store the generating rule and the used body tuples for every generated tuple during bottom up evaluation. While this makes the proof tree construction trivial and very fast, the additional memory usage is prohibitively large even for relatively small Datalog programs [32].

Another option is not to store any additional information during bottom up evaluation but to perform a brute force search for body tuples on the generated Database instance. However, for databases with millions of tuples this approach is very inefficient. Also no guarantees can be made for the minimality of proof trees.

To overcome these inefficiencies, Soufflé adds constant size proof annotations to tuples at evaulation time. While still allowing for a more efficient search, the annotations do not incur a prohibitively large memory overhead. The next two sections describe how those annotations are generated and how they are used for constructing proof trees.

3.2.2 Proof Annotations

Soufflé annotates every tuple with the identifier of the rule that was used for its generation and with the height of its proof tree. At the moment where a new tuple is generated during the evaluation, it is straightforward to know which rule is used for its generation. The height of the proof tree that corresponds to the currently generated tuple can be computed by taking the maximum of the heights of the proof trees of the body tuples and adding one. The height of facts is set to zero. Listing 3.3 shows the annotated version of the tuples of the graph reachability example introduced in Chapter 1. For example, the annotations for the tuple `path(1, 3)` tell us that the tuple was generated using rule `r2` and that the height of its proof tree is 2. This corresponds to the height of the tuple's proof tree displayed in Listing 3.2.

```
edge
original tuple      rule    height
1    2                -      0
2    3                -      0
3    4                -      0
4    5                -      0
5    4                -      0

path
original tuple      rule    height
1    2                1      1
2    3                1      1
3    4                1      1
4    5                1      1
5    4                1      1
1    3                2      2
2    4                2      2
3    5                2      2
4    4                2      2
5    5                2      2
1    4                2      3
2    5                2      3
1    5                2      4
```

Listing 3.3 – Proof annotations for path example

At the syntactical level, Soufflé treats the annotations as additional attributes, i.e. columns, of the relations. This is implemented by rewriting the rules before evaluation starts. A relation *R(x)* is transformed into *R(X, @rule, @height)*. A rule

$$r_i : R(X) :- R_1(X_1), ..., R_k(X_k).$$

is transformed into

$$r_i : R(X, i, max(@height_1, ..., @height_k)+1) :- R_1(X_1, \_, @height_1), ..., R_k(X_k, \_, @height_k).$$

3.2.3 Provenance Evaluation Strategy

Adding proof annotations requires some changes in the semantics of bottom up evaluation described in Section 3.1. As proof annotations are treated as additional attributes, using the standard fixpoint computation would generate all possible annotations for every original tuple. However, given that there are potentially infinitely many proof trees with different heights for every tuple, generating a new tuple for every possible height annotation would result in non-terminating executions. Consider for example proof trees for the tuple `path(4, 5)` in the running example. The cycle between node 4 and node 5 can be taken infinitely often to construct a path, yielding a possible proof tree with a bigger height every time.

To ensure termination, the adapted evaluation is required to add each original tuple only once, i.e. the proof annotations have to be unique for every original tuple. The key idea for ensuring uniqueness in Soufflé's provenance evaluation, is to check at the generation of a new tuple whether another tuple with the same original values already exists and if the height annotation of the newly generated tuple is smaller than the height annotation of the already existing tuple. If this is the case, the proof annotations are updated instead of adding the second tuple. If the height annotation of the newly generated tuple is greater than the height annotation of the already existing tuple, no updates will be performed. Since the first tuple found will have some finite height annotation k and tuples cannot have height annotations smaller than 0, there will be at most k updates for this tuple. This reasoning follows for every tuple and thus ensures the termination of the procedure. As a consequence of this approach, the unique final height annotation is minimal amongst all possible annotations and thus represents the height of a minimal height proof tree.

The adapted semantics can be formalized by defining the fixpoint computation over a special *provenance lattice* instead of the usual subset lattice. The *provenance lattice* is formed over pairs (I, h), where I is a set of generated tuples and $h : I -> \mathbb{N}_0$ is a function describing the smallest proof tree height discovered so far for every tuple in I. The order $\sqsubseteq$ of the lattice is defined by

$$(I_1, h_1) \sqsubseteq (I_2, h_2) \iff I1 \subseteq I2 \wedge \forall t \in I_1 : h_1(t) \geq h_2(t)$$

Simply put, a pair is *further up* in the lattice if it contains more tuples and the height annotations of those tuples are smaller. For more details, refer to [33].

Implementation

Soufflé's data structures are optimized to perform efficient updates of provenance annotations by updating the provenance attributes of the tuple that is already part of the internal data structure instead of deleting the existing tuple and inserting a completely new tuple.

Internally, in Soufflé every database tuple is stored in one or more specialized B-tree data structures [19]. In the B-tree, tuples are stored sorted by the values of their attributes. The order in which the attributes are taken into account for sorting is determined by a parameter of the data structure referred to as *index ordering*. Consider for example a relation with three attributes *A, B, C*. If the index ordering is e.g. *(B, C, A)*, tuples will first be compared by attribute *B*, then by attribute *C* and then by attribute *A*.

Every index ordering efficiently supports those searches that specify a set of attributes that form a prefix of the index ordering. For example index order *(B, C, A)* efficiently supports queries specifying no value, a value for *B*, values for *B* and *C*, or specifying values for all three attributes. In order to efficiently support search queries that specify different sets of attributes, several B-trees with different index orderings might be used to store the same relation [28].

Performing in-place updates of proof annotations is only a valid operation if the update preserves the tuples position in the index with regards to the index ordering. In Soufflé's provenance evaluation, this is assured by allowing only index orderings that have the height and rule attributes in the last two positions. This means that the resulting indexes only support searches specifying proof annotations if also all other values of the original tuple are specified. As proof annotations are never used for index lookups, neither during the evaluation nor during proof tree construction, this restriction does not pose a problem in Soufflé's current provenance component. However, as will be explained in Section 4.4, it presented a challenge for the implementation of the modified proof construction algorithm proposed in this thesis.

3.2.4 Proof Tree Generation

The proof tree generation component takes as input the annotated database of tuples generated in the bottom up evaluation and a specific tuple given by the user query. As explained above, the essential operation for constructing one level of a proof tree is to find for a tuple t that is annotated by rule r_i body tuples $t_1, ...t_k$, where $r_i : t :- t_1, ..., t_k$. This search will first be executed for the input, i.e. root tuple, and then applied recursively on all found body tuples until all body tuples are facts.

As the database was generated by a valid run of bottom up evaluation, it is ensured for every database tuple that corresponding body tuples exist is the database. From the fact that the

height of the tuple t was computed by adding 1 to the maximum height of the body tuples, one can conclude that every body tuple has a height less than the height of t. This is used to constrain the top down search.

For example if a user asks for a proof tree for the tuple `path(1, 4)`, first a look up for the tuple's height and rule annotation will be done. As can be seen in Listing 3.3, the tuple was generated by rule `r2` and has height 3.

Therefore a search for the body tuples will be performed with the following constraints:

$$\begin{aligned} path(x, z, , h_1), \\ edge(z, y, , h_2), \\ x = 1, \\ y = 4, \\ h_1 < 3, \\ h_2 < 3. \end{aligned}$$

In Soufflé, the search is implemented by a index nested loop join on the body relations. That is, there is a loop nest with one loop per body relation. The constraints are used to filter the tuples returned by the index lookup. The fewer tuples pass the filter, the fewer tuples have to be compared with tuples of the next relation of the join. Algorithm 1 provides a more detailed description of the computation.

The construction of the first level of the proof tree for tuple `path(1, 4)` works as follows. In the first step, an index is used to retrieve all path tuples starting in node 1. As can be checked in Listing 3.3, this will return the tuples `path(1, 2)`, `path(1, 3)`, `path(1, 4)` and `path(1, 5)`. In the next step, the tuples are filtered by the constraint $h_1 < 3$, which results in potential body tuples `path(1, 2)` and `path(1, 3)`, thereby removing branches from the search that will not yield results. Then, the first loop of the join iterates over these tuples and uses them as the basis for index lookups in the edge relation. The first lookup searches for edges that could we combined with potential body tuple `path(1, 2)`, i.e. the index is queried for edges from 2 to 4. As such an edge does not exist, the next potential body tuple `path(1, 3)` is taken into account and a lookup for edges from 3 to 4 is done. The result consists of one tuple, `edge(3, 4)`. After filtering the result by constraint $h_2 < 3$, tuples `path(1, 3)` and `edge(3, 4)` are added to the final set of body tuples and the construction of the proof tree level is done.

Algorithm 1 Search for body tuples

Input

I	Database instance
t	Original non-annotated tuple

Output

$t_1, ..., t_k$	body tuples generating t, i.e. $t :- t_1, ..., t_k$

1: Find $t' \in I$ such that $t' = (t, r, h)$ where r denotes rule

$$R(X_0, r, max(h_1, ..., h_k)) :- R_1(X_1, \_, h_1), \ ..., \ R_k(X_k, \_, h_k).$$

2: $S_1 := \{(t_1, \_, h_1) \in R_1 |\ t_1$ satisfies the constraints derivable from tuple values $t\}$
3: **for all** $(t_1, \_, h_1) \in S_1$ **do**
4: **if** $h_1 < h$ **then**
5: $S_2 := \{(t_2, \_, h_2) \in R_2 |\ t_2$ satisfies the constraints derivable from tuple values t and $t_1\}$
6: **for all** $(t_2, \_, h_2) \in S_2$ **do**
7: **if** $h_2 < h$ **then**
8: ...
9: $S_k := \{(t_k, \_, h_k) \in R_k |$
t_k satisfies the constraints derivable from tuple values t and $t_1, ..., t_{k-1}\}$
10: **for all** $(t_k, \_, h_k) \in S_k$ **do**
11: **if** $h_k < h$ **then**
12: **return** $t_1, t_2, ..., t_k$
13: **end if**
14: **end for**
15: **end if**
16: **end for**
17: **end if**
18: **end for**

4 Subtree-heights Provenance

In this chapter, we introduce *subtree-heights provenance*, an alternative way of computing proof trees for outputs of Datalog analyses in Soufflé. We present the framework for the newly introduced provenance algorithms and their implementation in Soufflé.

4.1 Key intuition

While the proof annotations used in Soufflé's provenance help in guiding the search for body tuples, there is still potential for further restricting the search space. Consider our running example of computing paths in a graph. The constraints enforce that only paths in the proof tree with heights smaller than the height of the original path are considered. However, given that the path body tuple is the only body tuple that is not a fact and all facts have height 0, it can be concluded that the path body tuple has the biggest height of all body tuples. By inverting the function that computes a tuple's height from the heights of its body tuples by taking the maximal body tuple height plus 1, we can conclude that the height of the path body tuple equals the height of the original tuple minus one. Therefore, stricter constraints could be used to find the body tuples in less steps:

$$
\begin{aligned}
path(x, z, , h_1),\\
edge(z, y, , h_2),\\
x = 1,\\
y = 4,\\
h_1 = 2,\\
h_2 = 0.
\end{aligned}
$$

To see how those constraints help reducing the search space, consider again the steps of the algorithm for computing the proof tree of tuple `path(1, 4)` illustrated at the end of the previous chapter. The difference will be that the constraint $h_1 = 2$ will restrict the result of

potential `path` body tuples to the tuple `path(1, 3)`, hence, cutting the branch `path(1, 2)`, avoiding the unnecessary further index lookups that are based on this branch and consequently reducing the runtime of the search. A similar reduction of the search space can be applied for arbitrary rules if, at proof construction time, the heights of the minimal proof trees of the body tuples is known.

4.1.1 Proof annotations

The core idea of the new provenance mechanism, subtree-heights provenance, is to store the height of the body tuples in addition to Soufflé's rule and height annotation during the evaluation. We will show now how to use these annotations to further reduce the search space during the top down proof tree construction.

Similarly to the other proof annotations, at a syntactical level, the subtree height annotations are treated as additional attributes and the rules are transformed accordingly before the start of the evaluation. A relation *R(x)* is transformed into

$$R(X, @rule, @height, @height_1, ..., @height_n)$$

where n denotes the maximal number of body clauses amongst all rules generating tuples of relation R. The number of additional height parameters is set to the maximum, as this ensures that all tuples of the relation have the same number of attributes.

A rule

$$\begin{aligned} r_i : R(X) :- \\ R_1(X_1), \\ ...., \\ R_k(X_k). \end{aligned}$$

is transformed into

$$\begin{aligned} r_i : R(X, i, max(@height_1, ..., @height_k) + 1, height_1, ..., @height_k, -1, ..., -1) :- \\ R_1(X_1, \_, @height_1, \_, ..., \_), \\ ..., \\ R_k(X_k, \_, @height_k, \_, ..., \_). \end{aligned}$$

The rule number and first height parameter of the relation are computed in the same way as for Soufflé's provenance. Then k additional height parameters are added, one per body clause of the rule. The ith additional height parameter represents the height of the ith body tuple.

```
edge
original tuple       rule    height
1    2                -       0
2    3                -       0
3    4                -       0
4    5                -       0
5    4                -       0

path
original tuple       rule    height  subheights
1    2                1       1       0    -1
2    3                1       1       0    -1
3    4                1       1       0    -1
4    5                1       1       0    -1
5    4                1       1       0    -1
1    3                2       2       1     0
2    4                2       2       1     0
3    5                2       2       1     0
4    4                2       2       1     0
5    5                2       2       1     0
1    4                2       3       2     0
2    5                2       3       2     0
1    5                2       4       3     0
```

Listing 4.1 – Extended proof annotations for path example

As above, let n denote the maximal number of body clauses amongst all rules generating tuples of relation R. The last $n-k$ attributes of all generated tuples are set to -1, as no body clauses exist for these positions and therefore the corresponding height parameters are undefined. The non-existing heights are encoded as -1, as this ensures that when height parameters of two tuples are compared, the missing height parameter corresponding to a non-exisitng subtree is smaller than the height parameter of any existing subtree.

Listing 4.1 shows the tuples of the path example with the additional annotations.

4.2 Subtree-heights Provenance Evaluation Strategy

While the height of the body tuples is easy to compute at evaluation time, similarly to Soufflé's provenance, it is necessary to refine the bottom up evaluation strategy. In particular, the update mechanism has to be adapted to take into account the additional subtree height annotations.

As outlined in Section 3.2.3, Soufflé's provenance evaluation is a fixpoint computation over pairs of instances and height metrics (I, h).

According to [33], Soufflé's provenance evaluation strategy can be adapted for different proof tree metrics by redefining the function h, as long as $h : I \to X$ satisfies the following conditions:

1. There is a partial order $\preceq$ on the codomain X that can be used for updating tuples with smaller annotations.
2. The function h can be compositionally computed from the body tuples, i.e. for a rule $t :- t_1, ..., t_n$, it can be computed as $h(t) = f(h(t_1), ..., h(t_n))$.
3. The function h is monotone and bounded. That is, for a rule $t :- t_1, ..., t_n$ and any $1 \leq i \leq n$ it holds that $h(t_i) \preceq h(t)$. Furthermore, there is a minimum value c such that $c \preceq h(t)$ for any tuple t.

For subtree-heights provenance evaluation, we define a new proof tree metric $\bar{h} : I \to X$ where

$$X = \left\{ (h_0, h_1, ..., h_n) \;\middle|\; n \in \mathbb{N}_0,\ h_0 \in \mathbb{N}_0,\ h_i \in \mathbb{N}_0 \cup \{-1\}\ \forall\ 1 \leq i \leq n \right\}$$

Let h denote the usual proof tree height metric that is used in Soufflé. The function $\bar{h}$ maps every database tuple t_0 that was generated by rule $t_0 :- t_1, ..., t_k$ in a relation with n additional height parameters to a tuple of height annotations $(h_0, h_1, ..., h_n)$, where $h_i = h(t_i)$ for all $0 \leq i \leq k$ and $h_i = -1$ for all $k < i \leq n$. That is, the first entry h_0 is defined as the minimal proof tree height of t_0 and the rest of the height annotations $h_1, ..., h_k$ describe the minimal proof tree heights of the body tuples and $h_{k+1}, ..., h_n$ are set to -1 encoding non-existent height parameters.

Let us now show that that $\bar{h}$ satisfies the required properties of proof tree metrics. The first requirement of having a partial order $\preceq$ on the codomain can be fulfilled by defining $\preceq$ as the lexicographical order over tuples of height annotations that compares the respective elements using $\leq$ over $\mathbb{N}_0 \cup \{-1\}$. The partial order properties of $\preceq$ follow trivially from the partial order properties of $\leq$.

Regarding compositionality, $\bar{h}$ can be computed as $\bar{h}(t) = (h(t), \bar{h}(t_1)[0], ..., \bar{h}(t_k)[0], -1, ..., -1)$, where $\bar{h}(t_i)[0]$ denotes the first entry of the height tuple of database tuple t_i for all $1 \leq i \leq k$.

For showing the monotonicity of $\preceq$, we need to show that $\bar{h}(t_i) \preceq \bar{h}(t)$ for an arbitrary i with $1 \leq i \leq k$. By the definition of $\preceq$, $\bar{h}(t_i)$ and $\bar{h}(t)$ are compared lexicographically. The first component of $\bar{h}(t_i)$ is $h(t_i)$ and the first component of $\bar{h}(t)$ is $h(t)$. From the fact that $h(t)$ is defined as the maximum of the body tuple heights plus one, it follows that $h(t_i) < h(t)$. Therefore, $\bar{h}(t_i) \preceq \bar{h}(t)$ holds independently of the values of the other elements in the tuples and monotonicity is shown. A lower bound is given by the tuple $(0, -1, ..., -1)$ consisting of a entry with value 0 followed by n entries with value -1, as the first height parameter has lower

bound 0 and the additional height parameters have lower bound −1. This shows that Soufflé provenance with subtree heights annotations is a valid provenance evaluation strategy.

4.3 Proof tree generation

As in Soufflé's provenance, the search for body tuples is implemented by an indexed nested loop join on the body relations. However, the sub-height constraints are now part of the index lookup instead of being enforced in an additional filter operation. Algorithm 2 outlines the steps for body tuple search using the additional annotations.

Algorithm 2 Search for body tuples using additional height annotations

Input

I	Database instance
t	Original non-annotated tuple

Output

$t_1, ..., t_k$	body tuples generating t, i.e. $t :- t_1, ..., t_k$

1: Find $t' \in I$ such that $t' = (t, r, h, h_1, ..., h_k)$ where r denotes rule

$$R(X_0, r, h, h_1, ..., h_k) :-$$
$$R_1(X_1, \_, h_1, \_, ..., \_),$$
$$...,$$
$$R_k(X_k, \_, h_k, \_, ..., \_).$$

2: $S_1 := \{(t_1, \_, h'_1, \_, ..., \_) \in R_1 \mid h'_1 = h_1$ and
t_1 satisfies the constraints derivable from tuple values $t\}$
3: **for all** $(t_1, \_, h'_1, \_, ..., \_) \in S_1$ **do**
4: $S_2 := \{(t_2, \_, h'_2, \_, ..., \_) \in R_2 \mid h'_2 = h_2$ and
t_2 satisfies the constraints derivable from tuple values t and $t_1\}$
5: **for all** $(t_2, \_, h'_2, \_, ..., \_) \in S_2$ **do**
6: ...
7: $S_k := \{(t_k, \_, h'_k, \_, ..., \_) \in R_k \mid h'_2 = h_2$ and
t_k satisfies the constraints derivable from tuple values t and $t_1, ..., t_{k-1}\}$
8: **for all** $(t_k, \_, h'_k, \_, ..., \_) \in S_k$ **do**
9: **return** $t_1, t_2, ..., t_k$
10: **end for**
11: **end for**
12: **end for**

With subtree-heights provenance, the construction of the first level of the proof tree for tuple `path(1, 4)` works as follows. In the first step, an index is used to retrieve all path tuples

starting in node 1 that have height 3. As can be checked in Listing 4.1, this will return only the tuple `path(1, 3)`. Then, the first loop of the join uses this tuple as the basis for an index lookup in the edge relation, that searches the index of the edge relation for edges from 3 to 4 with height 0. As the corresponding edge is found, tuples `path(1, 3)` and `edge(3, 4)` are added to the final set of body tuples and the construction of the proof tree level is done.

4.4 Implementation in Soufflé

The implementation of subtree heights provenance is publicly available at https://github.com/ssallinger/souffle.

Updates during evaluation are implemented in the same way as for Soufflé's original provenance. Every time a new tuple is inserted, a check is performed to see whether a tuple with the same entries for the original, non-provenance, attributes already exists. If this is the case, the new tuple is only inserted if the sequence of its height annotations is lexicographically smaller than the sequence of height annotations of the existing tuple.

As mentioned in Section 3.2.3, for efficiency reasons updates of proof annotations are not implemented by deleting the old tuple and inserting a new tuple but by making an in-place update in the data structure. This is only a valid operation if the update preserves the tuples position in the index with regards to the index ordering.

However, as explained above, subtree-height provenance relies on using the height parameter in index lookups. Such searches are only supported by indexes that order the height annotation before other attributes of the original tuple. Hence, there is a conflict between the different requirements on index orders.

In the implementation of subtree-heights provenance, this conflict of requirements is solved by constructing the indices that support lookups by height parameter only after the evaluation has terminated, by inserting all tuples of one of the other indexes.

It might be worth exploring the performance characteristics of an alternative approach that applies the height constraint only as a filter after doing the index lookup, similar to what is done in Soufflé's proof tree construction.

5 Experimental Evaluation

In this chapter, we present an experimental evaluation of subtree-heights provenance in Soufflé by measuring its impact on the performance of the bottom up evaluation and proof construction phases. In particular, we check the validity of the following hypotheses:

Hypothesis 1 Subtree-heights provenance reduces the number of index lookups at proof tree construction time and, hence, reduces the runtime of this phase.

Hypothesis 2 Subtree-heights provenance runtime overhead on the bottom up time is not prohibitive.

Hypothesis 3 Subtree-heights provenance does not incur in dramatic memory overhead, neither during evaluation nor proof construction.

Hypothesis 4 The above hypotheses also hold with Soufflé's magic set transformation.

In order to find evidence for those hypotheses, we compare subtree-heights provenance to the provenance computation as it is currently implemented in Soufflé (see Section 3.2) and, where applicable, to the execution of Soufflé without any provenance instrumentation.

5.1 Experimental Setup

The evaluation is done with two program analyses implemented in Soufflé. These analyses compute reachability between nodes, i.e. basic blocks, in a program's control flow graph (CFG) with different degrees of sensitivity. We chose these analyses since reachability between basic blocks are the basis for important classes of program analysis such as information flow. In particular, we focus on taint analyses which identify pairs of program locations (source, sink) that represent program locations where malicious users can inject data (source) and locations where such malicious data reaches potential sensitive information (sink). The analyses are run over a set of mature real world open source projects in Java and PHP. In the following sections,

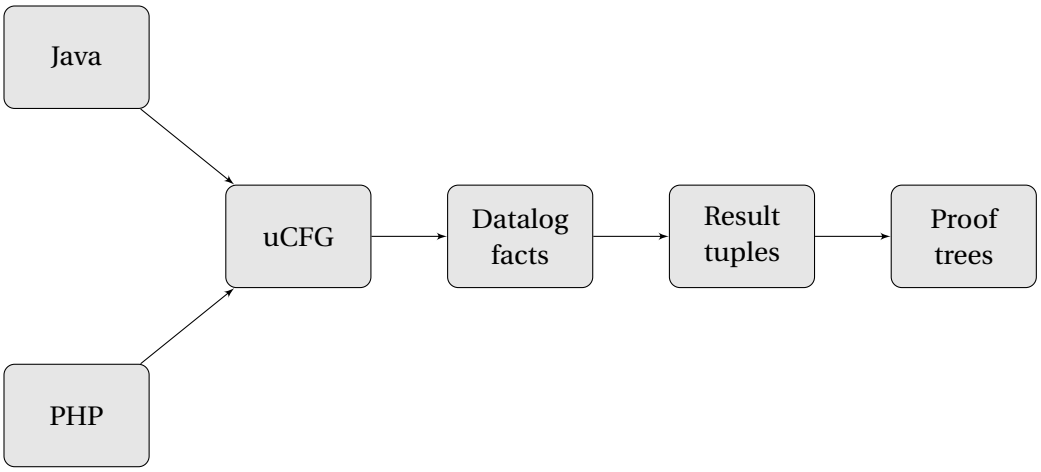

Figure 5.1 – Analysis pipeline

we give a detailed description of the analysis pipeline, the properties of the implemented analyses, and the input programs.

5.1.1 Analysis Pipeline

Figure 5.1 presents components of the analysis pipeline from the original program to the results of the analysis. In a first step, we retrieve the program's CFG. The implementation of this step relies on a transformation from programs written in Java and PHP to a common intermediate representation called *uCFG*, implemented in the current SonarSource taint analyzers. An open source version of *Sonar uCFG*, the library implementing the intermediate representation, is available at [26]. The analysis generates one uCFG per function consisting of its signature and basic blocks representing the intraprocedural CFG. A basic block follows similar design principles as the LLVM IR language [21] and consists of a sequence of instructions and a terminator indicating the reason for the change in control flow, i.e., the reason why this basic block ends. As usual, a terminator is a return or a jump to another basic block in which case it contains the destination basic block.

The next step is to compute a database of Datalog facts representing the CFG. The database consists of one or more Datalog relations containing the CFG's edges, i.e. pairs of nodes. Nodes are represented by globally unique names composed of the name of the function they belong to and a local identifier within the function. If the subsequently performed analysis does not need to distinguish intra- and interprocedural, all edges can be stored in the same relation. Otherwise, one relation will be created per edge type.

Datalog facts for intraprocedural edges can easily be generated from the uCFGs by inspecting for every basic block the references that it stores to its successors. Interprocedural edges are not modeled explicitly in uCFGs and require special treatment. Additional edges need to be added between basic blocks that contain function calls and basic blocks of the called

function. For the Datalog representation, if a basic block contains a function call, the basic block will be modeled by two nodes. The first node represents the part of the basic block containing all instructions up to and including the function call. The second part represents the continuation of the basic block containing instructions that will be executed after the function call returns. A Datalog fact is written for the call edge from the node corresponding to the first part of the basic block to the entry basic block of the called procedure. Furthermore, facts representing return edges from every return basic block of the called procedure to the continuation node of the calling basic block are added. In order to identify which function implementation to use for virtual calls, we rely on the call graph that is computed by the SonarSource analyzer based on the Variable Type Analysis algorithm [29]. Note uCFGs are usually not available for library functions. Therefore, calls to library functions are modelled by inserting an edge between the node modelling the first part of the basic block and the node modelling its continuation, hence, skipping the call to the unknown function.

In addition to the embedding of the CFG, two additional Datalog relations are created to represent the set of source nodes and the set of sink nodes for the taint analysis. A basic block is identified as source node if it contains a call to a function that takes input from the user and as sink node if it contains a call to a function which executes a potentially dangerous command, e.g. a direct execution of a SQL statement on a database.

Example

Consider for example the Java methods displayed in Listing 5.1. The function `getTheParameter` reads input from the user by parsing an HTTP request. The function `executeQuery` executes a SQL query. The goal of the analysis is to find out whether the HTTP parameter is used in the query. In the first step, the uCFGs are computed. The result is displayed in Listing 5.2. The function `getTheParameter` consists of two basic blocks, the function `executeQuery` consists of three basic blocks, each consisting of several instructions and a terminator. While return terminators specify the functions return value, jump terminators have a reference to the successors basic blocks. For example at the end of *basic block 4* of `executeQuery`, a jump to *basic block 3* of the same function is performed. Listing 5.3 shows the Datalog tuples that are generated for the interprocedural CFG in the next step of the pipeline. The corresponding graphical representation is shown in Fig. 5.2. The blue nodes belong to `getTheParameter`, the yellow ones to `executeQuery`. The names of the nodes consist of the name of the corresponding basic block followed by an identifier for the part of the basic block. As explained above, basic blocks are considered to be split into parts at function call instructions. Consider for example *basic block 4* of `executeQuery`. It is modelled by two nodes with identifiers `4^0` and `4^1`. Node `4^0` models the basic block up to the call to `getTheParameter` and has an outgoing edge to all entry blocks of the called function. Node `4^1` represents the continuation and has an incoming edge from every return block of the called function. Furthermore, it

```
1  public static String getTheParameter(String p, HttpServletRequest
     request, double random) {
2      if(random < 0.5)
3          return request.getParameter(p);
4      else
5          return "superSecurePwd";
6  }
7
8  public void executeQuery(String p, HttpServletRequest request) {
9      String param = getTheParameter(p, request, 0.0);
10     String sql = "INSERT INTO users (username, password) VALUES ('
         foo','" + param + "')";
11     try {
12         int count = statement.executeUpdate(sql,new int[]{1, 2});
13     } catch (java.sql.SQLException e) {
14         e.printStackTrace();
15     }
16 }
```

Listing 5.1 – Java functions to be represented in Datalog

has an outgoing edge to a node of the next basic block, *basic block 3*, modelling the jump instruction in the end of *basic block 4*.

In the next step, the analysis rules are applied to the input database in order to compute the output relation. In the case of reachability based taint analysis, the output relation contains pairs of nodes where the first node is a source, the second node is a sink and there exists a path between the two.

The last step of the analysis is to compute proof trees for the generated output tuples using Soufflé's built in provenance or the newly introduced subtree-heights provenance.

5.1.2 Analyses

Graph Reachability

Graph reachability is a basic component of many program analysis problems [25]. The first analysis that we consider, presented in Listing 5.4, computes graph reachability in its most basic form without distinguishing inter- and intraprocedural edges.

The analysis takes as input a relation describing the edges of the graph and two relations describing sources and sinks. It then computes the two relations `path` and `reachesSrcSink`. The `path` relation constitutes the core of the analysis. It describes that there is a path between

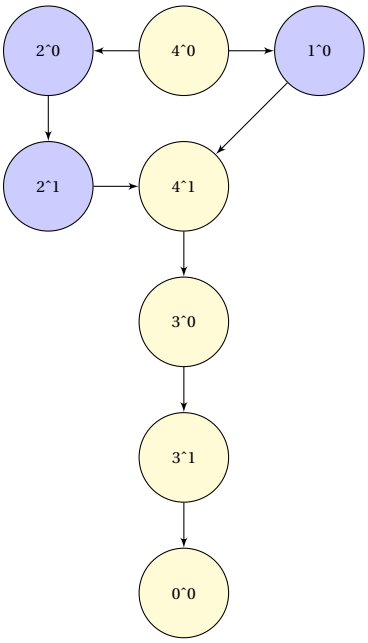

Figure 5.2 – CFG of the Java functions in Listing 5.1

```
1  //ucfgs for getTheParameter
2  bb2: //entry
3      %0 = getParameter[request, p]
4      return %0
5
6  bb1: //entry
7      return "superSecurePwd"
8
9  //ucfgs for executeQuery
10 bb4: //entry
11     %0 = call getTheParameter[p, request]
12     param = call __id[%0]
13     %1 = call __concat["INSERT INTO users (username, password)
14                         VALUES (\'foo\',\'", param]
15     %2 = call __concat[%1, "\')"]
16     sql = call __id[%2]
17     jump 3
18
19 bb3:
20     %3 = call __id [statement]
21     %4 = call executeUpdate[%3, sql]
22     jump 0
23
24 bb0:
25     return
```

Listing 5.2 – uCFGs of the Java functions in Listing 5.1

```
//intraprocedural edges
getTheParameter.2^0 getTheParameter.2^1
executeQuery.3^1    executeQuery.0^0
executeQuery.3^0    executeQuery.3^1
executeQuery.4^1    executeQuery.3^0

//interprocedural edges
executeQuery.4^0    getTheParameter.2^0
executeQuery.4^0    getTheParameter.1^0
getTheParameter.2^1 executeQuery.4^1
getTheParameter.1^0 executeQuery.4^1

//sources
getTheParameter.2^0

//sinks
executeQuery.3^0
```

Listing 5.3 – Datalog facts corresponding to the Java functions in Listing 5.1

```
.decl edge(x:symbol, y:symbol)
.input edge

.decl src(x:symbol)
.input src

.decl sink(x:symbol)
.input sink

.decl path(x:symbol, y:symbol)
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).

.decl reachesSrcSink(x:symbol, y:symbol)
reachesSrcSink(x, y):- src(x), sink(y), path(x, y).

.output reachesSrcSink
```

Listing 5.4 – Graph reachability analysis

```
1  .decl edge(x:symbol, y:symbol)
2  .input edge
3
4  .decl src(x:symbol)
5  .input src
6
7  .decl sink(x:symbol)
8  .input sink
9
10 .decl pathFromSrc(x:symbol, y:symbol)
11 pathFromSrc(x, y) :- edge(x, y).
12 pathFromSrc(x, y) :- src(x), pathFromSrc(x, z), edge(z, y).
13
14 .decl reachesSrcSink(x:symbol, y:symbol)
15 reachesSrcSink(x, y):- sink(y), pathFromSrc(x, y).
16
17 .output reachesSrcSink
```

Listing 5.5 – Efficient graph reachability analysis

two nodes if there is an edge between them or if their is a path from the first node to another node that is connected to the second node by an edge. The relation `reachesSourceSink` builds on the `path` relation by stating that a source is reachable from a sink if there is a path between them. Consider, for example, running the analysis on the Datalog facts from Listing 5.3. The `reachesSrcSink` relation will contain one tuple (`getTheParameter.2^0`, `executeQuery.3^0`).

For the experiments, we used a slight variation of the analysis that restricts paths to start in a source not in the rule for `reachesSrcSink`, but in the construction of the paths itself, hence, only computing paths starting in sources. The modified rules are displayed in Listing 5.5. Note that the two versions of the analysis are equivalent in the sense that they produce the same output. However, the modified version outperforms the original version in runtime and memory usage during the bottom up evaluation and allows us to scale this analysis to larger benchmarks where the first analysis would either time out or run out of memory (see Appendix A.1 for experimental data supporting this claim). The improvement is due to the fact that the number of source nodes is usually considerably smaller than the total number of nodes and that therefore the `pathFromSrc` relation will be considerably smaller than the `path` relation.

The analysis in Listing 5.5 is interprocedural, path-insensitive, 0-context-sensitive and partly flow-sensitive. It is path-insensitive as conditions on branches are not considered.

As the calling context is not considered in the analysis of procedure calls, the analysis is 0-context-sensitive. Both constraints might lead to false alarms, e.g. in situations where a function calls another function with a tainted parameter. If then a third function calls the same function, the result might be considered tainted even though the tainted value cannot reach the second call. The analysis is partly flow-sensitive as the order of basic blocks is taken into account, while the order of instructions within the same basic block is ignored.

Context-Free-Language Reachability

The second analysis is a more precise graph reachability analysis. We consider this analysis in our experiments in order to evaluate subtree-heights provenance in a more sophisticated scenario with more complex rules.

As stated in the previous section, the basic graph reachability analysis might yield false alarms because, in the construction of paths, the calling context is not taken into account and, hence, values are considered to flow from one caller of a function to the continuations of all the callers of the function. This source of imprecision can be overcome by considering Context-Free-Language reachability (CFL-reachability) [25].

In CFL-reachability, edges are associated with labels and nodes are only considered to be connected by a path if the concatenation of the labels of the edges on the path belongs to the considered context-free-language. The particular context-free language that forms the basis of the refinement is the language of matched parentheses defined over the alphabet $\Sigma = \{e\} \cup \{(_i \mid 1 \leq i \leq \text{CallSites}\} \cup \{)_i \mid 1 \leq i \leq \text{CallSites}\}$. Call sites are assumed to have unique identifiers. Every interprocedural edge describing a function call is associated with an opening parenthesis. The return edge corresponding to this call is labelled with the corresponding closing parenthesis. Intraprocedural edges are labelled by the constant symbol e. The language is defined by the following productions:

$$\text{matched} \rightarrow \text{matched}\ \ \text{matched} \tag{5.1}$$

$$\mid (_i\ \text{matched}\ )_i \qquad \text{for } 1 \leq i \leq \text{CallSites} \tag{5.2}$$

$$\mid e \tag{5.3}$$

The Datalog rules for CFL reachability analysis are presented in Listing 5.6. Lines 20 to 25 implement the productions of the language. The `path` relation directly corresponds to the *matched* non-terminal. The `onestep` relation is an intermediary step for computing paths produced by productions 5.2 and 5.3.

Restricting the set of valid paths to paths with well balanced parentheses makes sure that taint values at a call site can only flow along the continuation of the same call. However, in order to perform a taint analysis it is not sufficient to consider only matched paths as defined above between sources and sinks. The reason for this is that source and sink nodes do not have to appear on the same level of nested function calls. However, matching paths enforce that

there is a match for every parenthesis while the real goal is to prevent scenarios where to two parenthesis of different types would be matched. Still, matched paths can be used as a basic block to define the class of paths not containing mismatched parentheses as described in the following paragraphs.

To begin with, consider a scenario where a tainted value is read from a source at the outermost level of nested calls, say in function *main*. The tainted value is then passed to another function *f*, where it flows to a sink node. As there is an opening parenthesis for the call from *main* to *f* but no closing parenthesis for the corresponding return on the path from the source to the sink, the path is not well balanced even though it might be part of an execution of a program. This scenario can be modeled by allowing call edges that are not followed by return edges edges:

$$\begin{aligned} \text{forward\_realizable} \rightarrow\ & \text{forward\_realizable}\ \ \text{matched} && (5.4)\\ |\ & \text{forward\_realizable}\ (_i && \text{for } 1 \leq i \leq \text{CallSites} \quad (5.5)\\ |\ & \varepsilon && (5.6) \end{aligned}$$

As in this definition all return edges are part of matched paths, mismatches between parenthesis corresponding to different call sites are still prevented. In the Datalog program, these subpaths of well balanced paths are stored in relation `forward` that is computed by the rules in lines 35 to 37 of Listing 5.6.

Another scenario to consider occurs if the tainted value is read in a called function and flows to a sink in a calling function. This scenario can in turn be modeled by allowing return edges that are not preceded by call edges:

$$\begin{aligned} \text{backward\_realizable} \rightarrow\ & \text{matched}\ \ \text{backward\_realizable} && (5.7)\\ |\ &)_i\ \text{backward\_realizable} && \text{for } 1 \leq i \leq \text{CallSites} \quad (5.8)\\ |\ & \varepsilon && (5.9) \end{aligned}$$

In this case, all call edges are part of matched paths, such that mismatches between parenthesis corresponding to different call sites are still prevented. In the Datalog program, these subpaths of well balanced paths are stored in relation `backward` that is computed by the rules in lines 40 to 42 of Listing 5.6.

What remains to be solved is a third scenario where a taint source occurs in a called function and is then passed to another function where the taint flows to a sink. This can be allowed by considering paths that consist of a backwards reachable path, i.e. a path containing arbitrary returns, followed by a forwards reachable paths, i.e. a path containing arbitrary calls. In this case, parentheses mismatches are prevented by the fact that all closing parentheses that do not have a matching partner occur before all opening parentheses that do not have a partner. That is exactly how taint paths are computed in the `reachesSrcSink` rule in line 45 of Listing 5.6.

```
.decl edge(x:symbol, y:symbol)
.input edge

.decl continuation(x:symbol, y:symbol)
.input continuation

.decl call(x:symbol, y:symbol)
.input call

.decl ret(x:symbol, y:symbol)
.input ret

.decl src(x:symbol)
.input src

.decl sink(x:symbol)
.input sink

.decl path(x:symbol, y:symbol)
path(x, y) :- onestep(x,y).
path(x, y) :- onestep(x,z), path(z, y).

.decl onestep(x:symbol, y:symbol)
onestep(x,y) :- edge(x,y).
onestep(x,y) :- continuation(x, y), call(x, entry), ret(exit,
  y), path(entry, exit).

.decl node(x:symbol)
node(s) :- edge(s, _).
node(s) :- edge(_, s).
node(s) :- call(s, _).
node(s) :- call(_, s).
node(s) :- ret(_, s).

.decl forward(x:symbol, y:symbol)
forward(s, s):- node(s).
forward(s, y) :- forward(s, x), path(x, y).
forward(s, y) :- forward(s, x), call(x, y).

.decl backward(x:symbol, y:symbol)
backward(s, s) :- node(s).
backward(x, s) :- path(x, y), backward(y,s).
backward(x, s) :- ret(x, y), backward(y, s).

.decl reachesSrcSink(x:symbol, y:symbol)
reachesSrcSink(x, z) :- src(x), backward(x, y), forward(y, z)
  , sink(z).

.output reachesSrcSink
```

Listing 5.6 – CFL reachability analysis

ID	Name	lines of code	number of functions
1	Apache Tika	76843	5723
2	Facebook Buck	300655	25192
10	Spark	6127	838
17	docker-maven-plugin	16531	1981
26	AisLib application framework	12191	1188
29	Apache Abdera	49214	6922
33	Apache Commons Configuration	28021	2887
38	Apache Empire-db	48947	4917
39	Apache Jackrabbit	273574	23796
41	Maven Release	10984	824
43	Apache Tobago	41766	3960
45	Apache Pluto	23607	2436
46	Wicket Parent	137186	12964
79	Stapler Parent	8108	871
80	Maven hpi Plugin	1621	116
85	Restcomm Sip Servlets	43107	4085
103	Pippo Parent	16327	1935
110	Apache Tomcat-7	282648	27095
111	Apache Tomcat-8	328848	30159
112	Apache Tomcat-9	259455	24639

Table 5.1 – Java input programs

5.1.3 Benchmarks

For the experiments, we ran the analyses on a collection of big open-source projects in Java and PHP. Table 5.1 and Table 5.2 list the names of the input projects and metrics on the lines of code in the project, the number of functions and the number of Datalog input facts, for Java and PHP respectively. As the content of the projects is not of relevance in our evaluation, we omit a description and only refer to the projects by their benchmark IDs in the following sections.

5.2 Results

The experiments were performed in a machine with four Intel Xeon 2.20GHz CPUs and 26 GB of memory, running Ubuntu 18.04. In all experiments Soufflé, is used to generate C++ code parallelized for four cores that was then compiled with GCC 7.4.0. We only report results for those project where at least 50 outputs are generated at evaluation time in order to have a sufficiently large set of trees on which to evaluate the performance of proof tree construction.

ID	Name	lines of code	number of functions
0	Jeedom	36605	1631
2	UBUGraph	9021	538
7	CakePHP	63868	5150
11	CodeIgniter 2	28298	1566
12	CodeIgniter 3	28298	1566
13	CodeIgniter 4	30624	2183
14	Composer Dependency Manager for PHP	28009	1927
18	Drupal	239249	21448
26	Grav	33216	3153
30	Joomla CMS	253024	10007
36	Magento 2	532722	41725
37	Matomo	201350	14342
42	osCommerce Online Merchant v2.x	38226	1742
43	osCommerce Online Merchant v3.x	36066	1784
44	Phabricator	453360	35869
47	phpMyFAQ	71551	1138
52	phpMyAdmin	96539	3085
56	PimCore	180891	14287
58	PrestaShop Scalable eCommerce Solution	183226	8788
59	ProjectSend	71001	12329
61	Shopware	268650	20119
67	Typo3	413264	22929

Table 5.2 – PHP input programs

	Bottom up time (s)				Bottom up maximum RSS (KB)			
	no	exp	sH	sH/exp	no	exp	sH	sH/exp
1	1.48e-01	1.56e-01	1.55e-01	9.97e-01	28784	28968	31476	1.09e+00
2	1.01e+00	8.55e-01	9.66e-01	1.13e+00	115628	115768	132252	1.14e+00
10	1.99e-02	1.93e-02	2.34e-02	1.21e+00	6164	6452	7340	1.14e+00
17	8.54e-02	8.18e-02	1.01e-01	1.23e+00	12180	12488	15416	1.23e+00
26	1.61e-02	1.36e-02	1.59e-02	1.17e+00	7512	7720	8004	1.04e+00
29	4.44e-01	4.14e-01	6.49e-01	1.57e+00	28696	29036	41448	1.43e+00
33	7.10e-02	6.77e-02	8.61e-02	1.27e+00	13412	13688	15436	1.13e+00
38	1.50e-01	1.27e-01	1.54e-01	1.21e+00	20092	20368	23280	1.14e+00
41	1.64e-02	1.52e-02	1.85e-02	1.21e+00	7856	8132	8712	1.07e+00
43	1.31e-01	1.24e-01	1.63e-01	1.32e+00	17984	18372	21976	1.20e+00
45	6.07e-02	5.80e-02	6.91e-02	1.19e+00	11728	11860	14152	1.19e+00
46	6.51e+00	5.88e+00	8.73e+00	1.49e+00	156888	157084	293356	1.87e+00
79	8.44e-02	8.21e-02	1.05e-01	1.28e+00	9080	9308	12472	1.34e+00
80	3.56e-03	3.41e-03	4.10e-03	1.20e+00	5680	5788	5736	9.91e-01
85	9.35e-02	8.91e-02	1.11e-01	1.25e+00	16036	16240	19132	1.18e+00
103	2.36e-01	2.25e-01	3.48e-01	1.55e+00	15144	15456	23524	1.52e+00

Table 5.3 – Bottom up time and maximum RSS for running graph reachability analysis on Java benchmarks without provenance (no), with Soufflé's explain provenance (exp), and with subtree-heights provenance (sH).

All of the presented times are in seconds, memory is measured in KB. For memory usage the maximum resident set size (RSS), i.e. the maximum amount of main memory the process occupies at some point during its lifetime, is reported.

5.2.1 Bottom Up Evaluation using Subtree-heights Provenance

In this section, we evaluate the performance of bottom up evaluation with the additional instrumentation for subtree-heights provenance in order to gather evidence for Hypotheses 2 and 3.

Table 5.3 and Table 5.4 show a comparison of time and memory usage during bottom up phase for reachability analysis run on Java and PHP projects respectively. For time as well as memory the first three columns refer to executions without provenance (no), with Soufflé's explain provenance (exp), and with subtree-heights provenance (sH). The fourth column describes the ratio between measured values for executions with subtree-heights provenance and executions with Soufflé's provenance. On average, for Java as well as PHP benchmarks subtree-heights provenance instrumentation results in a 27% runtime overhead compared to Soufflé's provenance. The average memory overhead is 23% for Java projects and 33% for PHP projects.

	Bottom up time (s)				Bottom up maximum RSS (KB)			
	no	exp	sH	sH/exp	no	exp	sH	sH/exp
2	2.08e-01	1.97e-01	2.76e-01	1.40e+00	11512	11708	17680	1.51e+00
7	7.37e-02	7.16e-02	7.79e-02	1.09e+00	15352	15784	17204	1.09e+00
11	1.37e-01	1.42e-01	1.75e-01	1.23e+00	12656	13032	17060	1.31e+00
12	1.34e-01	1.47e-01	1.72e-01	1.18e+00	12876	13212	16992	1.29e+00
13	5.37e-02	5.36e-02	6.78e-02	1.27e+00	9932	10204	11628	1.14e+00
14	5.34e-02	5.49e-02	5.66e-02	1.03e+00	10624	10928	11924	1.09e+00
18	5.03e-01	5.27e-01	6.49e-01	1.23e+00	53168	53476	61480	1.15e+00
26	3.42e-02	3.28e-02	4.03e-02	1.23e+00	10396	10612	11376	1.07e+00
36	8.43e-01	7.80e-01	9.37e-01	1.20e+00	110196	110412	119844	1.09e+00
44	2.74e+00	2.69e+00	3.54e+00	1.32e+00	121388	121852	167140	1.37e+00
47	2.40e+00	2.40e+00	3.42e+00	1.43e+00	66428	67008	126368	1.89e+00
56	1.23e+00	1.29e+00	1.73e+00	1.34e+00	60868	61076	87012	1.42e+00
58	1.32e+00	1.34e+00	1.83e+00	1.37e+00	71040	71296	96864	1.36e+00
59	3.51e+00	3.48e+00	4.92e+00	1.41e+00	93544	93816	181380	1.93e+00
61	4.98e-01	5.13e-01	6.73e-01	1.31e+00	54328	54704	64752	1.18e+00

Table 5.4 – Bottom up time and maximum RSS for running graph reachability analysis on PHP benchmarks without provenance (no), with Soufflé's explain provenance (exp), and with subtree-heights provenance (sH).

The same measurements for CFL reachability are shown in Table 5.5 and Table 5.6. The runtime overhead for Java projects is on average 40%, for PHP projects 50%. Regarding memory usage, the average overhead is 40% for Java and 44% for PHP projects. The size of the memory overhead is determined by the ratio between the number of original columns and the number of additional height parameters. As explained in Section 4.4, the number of height parameters is determined by the maximal number of premises of all rules of a relation. For CFL reachability, this ratio is higher than for graph reachability which might explain the higher memory overhead. The runtime overhead is mostly caused by the additional updates of the provenance columns. In subtree-heights provenance, if a tuple's height is updated, also the sub-heights of all the tuples that where generated from this tuple potentially need to be updated. The more sub-height parameters a tuple has, the more likely it is that one of the heights needs to be updated which could explain the difference in overheads between graph and CFL reachability.

Hypothesis 2 claims that the runtime overhead on bottom up time is not prohibitive. While the runtime overhead might seem substantial in relative terms, for graph reachability the absolute times for bottom up evaluation are in a range of maximum a few seconds even for big benchmarks. Therefore, the overhead is a fraction of a second in most cases which seems very supportable for this analysis.

	Bottom up time (s)				Bottom up maximum RSS (KB)			
	no	exp	sH	sH/exp	no	exp	sH	sH/exp
26	1.25e+00	1.26e+00	1.80e+00	1.43e+00	19084	19260	25468	1.32e+00
39	9.10e+01	9.23e+01	1.29e+02	1.40e+00	623080	623300	885236	1.42e+00
103	2.66e+00	2.66e+00	3.66e+00	1.38e+00	32092	32416	43992	1.36e+00
110	1.20e+02	1.21e+02	1.70e+02	1.41e+00	655000	655232	936780	1.43e+00
111	1.54e+02	1.56e+02	2.16e+02	1.38e+00	698204	698484	998104	1.43e+00
112	1.33e+02	1.36e+02	1.90e+02	1.40e+00	592120	592412	845564	1.43e+00

Table 5.5 – Bottom up time and maximum RSS for running CFL reachability analysis on Java benchmarks without provenance (no), with Soufflé's explain provenance (exp), and with subtree-heights provenance (sH).

	Bottom up time (s)				Bottom up maximum RSS (KB)			
	no	exp	sH	sH/exp	no	exp	sH	sH/exp
0	5.34e+02	5.39e+02	7.68e+02	1.42e+00	749968	750296	1094388	1.46e+00
2	4.06e+01	4.18e+01	6.90e+01	1.65e+00	46944	47228	68744	1.46e+00
7	6.68e+00	6.83e+00	1.00e+01	1.47e+00	69884	69968	98008	1.40e+00
11	4.55e+00	4.56e+00	6.89e+00	1.51e+00	41460	41648	58060	1.39e+00
12	4.46e+00	4.54e+00	6.89e+00	1.52e+00	41404	41880	58272	1.39e+00
18	6.61e+01	6.68e+01	9.77e+01	1.46e+00	334868	335112	477924	1.43e+00
30	2.02e+02	2.05e+02	3.17e+02	1.55e+00	651876	652108	939952	1.44e+00
37	2.80e+01	2.85e+01	3.99e+01	1.40e+00	214536	214864	305856	1.42e+00
42	3.25e+02	3.28e+02	4.58e+02	1.40e+00	693040	693404	1012204	1.46e+00
43	1.92e+02	1.95e+02	3.02e+02	1.55e+00	397252	397600	580476	1.46e+00
47	6.91e+01	7.06e+01	1.02e+02	1.45e+00	129344	129564	187316	1.45e+00
52	2.93e+02	2.98e+02	4.47e+02	1.50e+00	419944	420308	618412	1.47e+00
56	8.07e+01	8.17e+01	1.24e+02	1.51e+00	284976	285176	416264	1.46e+00
58	4.26e+02	4.32e+02	6.82e+02	1.58e+00	533864	534308	773988	1.45e+00
59	4.71e+01	4.80e+01	6.89e+01	1.43e+00	115568	115772	169932	1.47e+00
67	2.53e+02	2.58e+02	4.00e+02	1.55e+00	839768	839888	1221684	1.45e+00

Table 5.6 – Bottom up time and maximum RSS for running CFL reachability analysis on PHP benchmarks without provenance (no), with Soufflé's explain provenance (exp), and with subtree-heights provenance (sH).

For CFL reachability, the absolute runtime overhead is in the range of up to some minutes. See for example PHP benchmark 58 where the overall runtime for bottom up evaluation increases from roughly 7 to roughly 11 minutes. While the overhead might be noticeable, it can be concluded that for the considered analyses it is not prohibitive.

Similarly, for Hypothesis 3 concerning memory overhead during bottom up time, we conclude that it is significant but not dramatic. The absolute differences in memory usage range from a few MB to at most a few hundred MB.

5.2.2 Proof Tree Construction using Subtree-heigths Provenance

This section presents performance characteristics of proof tree construction using subtree-heights provenance. In the evaluation, we measured the performance for constructing proof trees of all outputs of the analysis. While the provenance component of Soufflé mostly targets interactive queries that compute one proof tree at a time, we compute all of them in order to capture the overall performance. Furthermore, computing all proof trees is indeed an operation that might be needed for certain applications which rank analysis alarms based on user feedback [24] described in Chapter 2.

For the graph reachability analysis, Table 5.7 and Table 5.8 provide for every benchmark the number of output tuples, i.e. the number of computed trees, the average number of nodes per tree and the total number of nodes in the tree. The number of outputs varies strongly amongst the benchmarks. The numbers range from a few dozens to ten thousands of trees. For the Java benchmarks, the average number of nodes per tree is 90, for PHP benchmarks 63. The number of nodes is equal for Soufflé's provenance and subtree-heights provenance. As minimal height proof trees are not unique, this is not necessarily the case if more complex analyses are taken into account. For this simple analysis, it makes sense however, as all proof trees with the same height also have the same shape because every internal level of the proof tree except the first and the last one has exactly three nodes, corresponding to the three body tuples in the recursive rule for constructing a path.

The descriptions of the trees for CFL reachability are displayed in Table 5.9 and Table 5.10. As expected, the higher precision of CFL reachability leads to a considerably lower number of alarms, i.e. outputs. For the majority of Java benchmarks that are considered for graph reachability, CFL reachability does not yield any outputs at all. As mentioned above, benchmarks with less than 50 outputs are not considered in the discussion. For Java the maximal number of outputs is a 385. For PHP there are still benchmarks with thousands of outputs. The average number of node per tree is 41 for Java and 55 for PHP projects.

For reachability, Table 5.11 and Table 5.12 provide evidence for Hypothesis 1 that claims that subtree-heights provenance greatly reduce the number of index accesses during proof tree construction and therefore significantly reduce the runtime.

		exp		sH	
	# trees	# nodes	avg # nodes	# nodes	avg # nodes
1	1.79e+02	1.56e+04	8.69e+01	1.56e+04	8.69e+01
2	1.37e+03	1.18e+05	8.61e+01	1.18e+05	8.61e+01
10	1.62e+02	1.80e+04	1.11e+02	1.80e+04	1.11e+02
17	5.46e+02	7.00e+04	1.28e+02	7.00e+04	1.28e+02
26	6.20e+01	2.65e+03	4.28e+01	2.65e+03	4.28e+01
29	1.32e+03	1.49e+05	1.13e+02	1.49e+05	1.13e+02
33	8.40e+01	5.73e+03	6.82e+01	5.73e+03	6.82e+01
38	8.70e+01	5.64e+03	6.49e+01	5.64e+03	6.49e+01
41	7.40e+01	8.25e+03	1.12e+02	8.25e+03	1.12e+02
43	1.05e+02	1.15e+04	1.10e+02	1.15e+04	1.10e+02
45	6.80e+01	7.61e+03	1.12e+02	7.61e+03	1.12e+02
46	1.08e+04	6.94e+05	6.43e+01	6.94e+05	6.43e+01
79	2.50e+02	2.29e+04	9.16e+01	2.29e+04	9.16e+01
80	5.20e+01	6.00e+03	1.15e+02	6.00e+03	1.15e+02
85	5.60e+01	4.63e+03	8.26e+01	4.63e+03	8.26e+01
103	1.68e+03	7.59e+04	4.51e+01	7.59e+04	4.51e+01

Table 5.7 – Number of trees and average number of nodes per tree for running graph reachability analysis on Java benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

		exp		sH	
	# trees	# nodes	avg # nodes	# nodes	avg # nodes
2	6.72e+02	5.35e+04	7.96e+01	5.35e+04	7.96e+01
7	4.92e+02	3.12e+04	6.35e+01	3.12e+04	6.35e+01
11	2.62e+03	1.38e+05	5.28e+01	1.38e+05	5.28e+01
12	2.62e+03	1.38e+05	5.28e+01	1.38e+05	5.28e+01
13	9.67e+02	7.17e+04	7.41e+01	7.17e+04	7.41e+01
14	1.16e+03	1.17e+05	1.01e+02	1.17e+05	1.01e+02
18	2.71e+03	1.35e+05	4.98e+01	1.35e+05	4.98e+01
26	2.23e+02	8.48e+03	3.80e+01	8.48e+03	3.80e+01
36	7.10e+01	2.99e+03	4.21e+01	2.99e+03	4.21e+01
44	3.81e+03	2.80e+05	7.36e+01	2.80e+05	7.36e+01
47	8.83e+03	5.93e+05	6.72e+01	5.93e+05	6.72e+01
56	1.25e+04	7.54e+05	6.01e+01	7.54e+05	6.01e+01
58	1.57e+04	1.42e+06	9.05e+01	1.42e+06	9.05e+01
59	6.99e+04	3.35e+06	4.80e+01	3.35e+06	4.80e+01
61	1.20e+03	7.08e+04	5.90e+01	7.08e+04	5.90e+01

Table 5.8 – Number of trees and average number of nodes per tree for running graph reachability analysis on PHP benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

		exp		sH	
	# trees	# nodes	avg # nodes	# nodes	avg # nodes
26	6.00e+01	1.51e+03	2.52e+01	1.50e+03	2.50e+01
39	6.20e+01	1.34e+03	2.15e+01	1.31e+03	2.12e+01
103	1.48e+02	1.20e+04	8.12e+01	1.17e+04	7.88e+01
110	3.85e+02	1.74e+04	4.52e+01	1.65e+04	4.28e+01
111	2.71e+02	9.96e+03	3.67e+01	9.70e+03	3.58e+01
112	3.03e+02	1.10e+04	3.62e+01	1.06e+04	3.50e+01

Table 5.9 – Number of trees and average number of nodes per tree for running CFL reachability analysis on Java benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

		exp		sH	
	# trees	# nodes	avg # nodes	# nodes	avg # nodes
0	2.14e+03	1.55e+05	7.25e+01	1.45e+05	6.81e+01
2	5.50e+01	3.34e+03	6.08e+01	2.76e+03	5.01e+01
7	5.80e+01	1.62e+03	2.80e+01	1.57e+03	2.70e+01
11	7.20e+01	1.38e+03	1.91e+01	1.37e+03	1.90e+01
12	7.20e+01	1.38e+03	1.91e+01	1.37e+03	1.90e+01
18	2.92e+02	2.06e+04	7.07e+01	2.04e+04	6.99e+01
30	2.83e+02	1.08e+04	3.83e+01	1.06e+04	3.76e+01
37	5.61e+02	3.25e+04	5.80e+01	3.16e+04	5.64e+01
42	3.22e+04	4.32e+06	1.34e+02	4.07e+06	1.26e+02
43	2.28e+04	1.63e+06	7.16e+01	1.55e+06	6.82e+01
47	1.86e+02	3.97e+03	2.14e+01	3.86e+03	2.07e+01
52	4.40e+03	3.40e+05	7.72e+01	3.21e+05	7.31e+01
56	1.33e+02	4.38e+03	3.29e+01	4.18e+03	3.14e+01
58	3.63e+03	3.09e+05	8.52e+01	3.03e+05	8.35e+01
59	2.21e+03	1.29e+05	5.82e+01	1.24e+05	5.61e+01
67	1.41e+03	4.99e+04	3.55e+01	4.86e+04	3.45e+01

Table 5.10 – Number of trees and average number of nodes per tree for running CFL reachability analysis on PHP benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

	Proof tree construction time (s)			Number of index lookups		
	exp	sH	sH/exp	exp	sH	sH/exp
1	1.15e+01	1.27e+00	1.11e-01	7.36e+07	4.03e+06	5.48e-02
2	3.48e+02	3.74e+01	1.08e-01	1.82e+09	9.47e+07	5.21e-02
10	5.57e-01	1.29e-01	2.32e-01	3.63e+06	1.52e+05	4.18e-02
17	7.32e+00	1.20e+00	1.64e-01	4.65e+07	3.05e+06	6.55e-02
26	1.61e-02	1.45e-02	8.98e-01	3.47e+04	5.17e+03	1.49e-01
29	5.23e+01	6.37e+00	1.22e-01	3.20e+08	1.87e+07	5.85e-02
33	1.41e+00	2.23e-01	1.58e-01	9.91e+06	7.37e+05	7.43e-02
38	3.20e+00	4.97e-01	1.55e-01	2.27e+07	1.55e+06	6.82e-02
41	3.74e-01	7.31e-02	1.95e-01	2.35e+06	1.12e+05	4.75e-02
43	3.20e+00	3.76e-01	1.17e-01	2.46e+07	1.07e+06	4.35e-02
45	2.26e-01	6.13e-02	2.71e-01	1.37e+06	6.70e+04	4.90e-02
46	6.65e+02	1.18e+02	1.78e-01	3.67e+09	3.30e+08	8.99e-02
79	1.07e+00	2.26e-01	2.11e-01	6.58e+06	4.01e+05	6.09e-02
80	1.63e-01	4.16e-02	2.55e-01	1.32e+06	5.41e+04	4.10e-02
85	1.59e+00	1.93e-01	1.21e-01	1.21e+07	6.27e+05	5.18e-02
103	9.75e+00	1.97e+00	2.02e-01	6.44e+07	6.36e+06	9.87e-02

Table 5.11 – Time for constructing proof trees and number of index accesses for all outputs for running graph reachability analysis on Java benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

For Java as well as PHP benchmarks the reduction in index accesses is more than 90% and the reduction in runtime is almost 80%. Given that the proof tree construction is a big proportion of the overall runtime, this has a great impact in practice. While in bottom up evaluation we observe runtimes up to a few seconds at most, proof tree construction can take up to a few minutes for some of the considered benchmarks.

For CFL reachability the results are shown in Table 5.13 and Table 5.14. For this analysis, the number of index accesses is reduced by more than 50% for Java as well as PHP projects. The runtime improvement is 6% for Java projects and 15% for PHP projects. It seems that the effect of the reduction of index lookups on the runtime is less clear for this analysis. Further investigations are needed to understand why this is the case.

For Hypothesis 1, we conclude that there is a significant reduction of index accesses for both considered analyses. While this yields a dramatic speed up for graph reachability, the runtime of proof tree construction for CFL reachability is only slightly reduced.

Finally, we investigated some potential overheads that might be introduced by the way subtree-heights provenance is currently implemented. As explained in Section 4.4, after the Datalog evaluation finishes, specific indexes are added for the search of body tuples during proof tree construction. This results in an overhead in runtime as well as memory consumption.

	Proof tree construction time (s)			Number of index lookups		
	exp	sH	sH/exp	exp	sH	sH/exp
2	2.04e+00	4.65e-01	2.28e-01	1.45e+07	8.26e+05	5.70e-02
7	3.21e+00	6.62e-01	2.06e-01	1.74e+07	1.89e+06	1.08e-01
11	8.38e+00	1.91e+00	2.28e-01	5.15e+07	4.86e+06	9.45e-02
12	8.52e+00	1.93e+00	2.26e-01	5.15e+07	4.86e+06	9.45e-02
13	3.08e+00	7.10e-01	2.31e-01	1.77e+07	1.32e+06	7.46e-02
14	1.05e+01	1.64e+00	1.55e-01	7.09e+07	3.88e+06	5.47e-02
18	1.08e+02	2.19e+01	2.03e-01	4.86e+08	5.59e+07	1.15e-01
26	1.00e+00	2.26e-01	2.26e-01	6.85e+06	7.05e+05	1.03e-01
36	2.57e+00	5.00e-01	1.95e-01	1.40e+07	1.32e+06	9.44e-02
44	4.60e+02	5.72e+01	1.24e-01	1.97e+09	1.21e+08	6.17e-02
47	1.16e+02	2.04e+01	1.76e-01	8.62e+08	7.05e+07	8.18e-02
56	3.55e+02	7.70e+01	2.17e-01	1.65e+09	2.00e+08	1.21e-01
58	1.05e+02	1.77e+01	1.68e-01	6.39e+08	3.55e+07	5.56e-02
59	2.37e+02	5.30e+01	2.23e-01	1.55e+09	1.62e+08	1.04e-01
61	2.82e+01	5.72e+00	2.03e-01	1.89e+08	1.70e+07	8.99e-02

Table 5.12 – Time for constructing proof trees and number of index accesses for all outputs for running graph reachability analysis on PHP benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

	Proof tree construction time (s)			Number of index lookups		
	exp	sH	sH/exp	exp	sH	sH/exp
26	2.12e-02	2.09e-02	9.86e-01	8.13e+03	5.50e+03	6.77e-01
39	2.21e-02	2.05e-02	9.29e-01	8.10e+03	5.53e+03	6.82e-01
103	1.50e-01	1.45e-01	9.63e-01	6.62e+04	3.83e+04	5.78e-01
110	2.67e-01	2.40e-01	8.98e-01	1.49e+05	8.11e+04	5.45e-01
111	1.66e-01	1.53e-01	9.19e-01	9.74e+04	5.33e+04	5.47e-01
112	1.91e-01	1.77e-01	9.27e-01	1.18e+05	5.99e+04	5.09e-01

Table 5.13 – Time for constructing proof trees and number of index accesses for all outputs for running CFL reachability analysis on Java benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

	Proof tree construction time (s)			Number of index lookups		
	exp	sH	sH/exp	exp	sH	sH/exp
0	4.33e+00	2.85e+00	6.59e-01	3.02e+06	1.29e+06	4.29e-01
2	5.18e-02	3.89e-02	7.51e-01	2.49e+04	9.82e+03	3.95e-01
7	2.97e-02	2.67e-02	8.96e-01	2.18e+04	1.04e+04	4.77e-01
11	2.09e-02	2.02e-02	9.65e-01	7.81e+03	5.53e+03	7.08e-01
12	2.06e-02	2.04e-02	9.91e-01	7.81e+03	5.53e+03	7.08e-01
18	3.22e-01	2.99e-01	9.28e-01	1.99e+05	8.84e+04	4.45e-01
30	1.81e-01	1.73e-01	9.56e-01	9.84e+04	5.90e+04	6.00e-01
37	7.38e-01	5.25e-01	7.11e-01	6.84e+05	2.39e+05	3.50e-01
42	6.52e+01	5.53e+01	8.49e-01	3.36e+07	1.74e+07	5.18e-01
43	2.30e+01	2.13e+01	9.26e-01	9.93e+06	6.03e+06	6.07e-01
47	5.92e-02	5.65e-02	9.55e-01	2.19e+04	1.42e+04	6.48e-01
52	8.01e+00	6.08e+00	7.60e-01	5.94e+06	2.91e+06	4.90e-01
56	6.37e-02	5.90e-02	9.26e-01	3.36e+04	1.70e+04	5.05e-01
58	5.86e+00	4.94e+00	8.44e-01	2.94e+06	1.47e+06	5.02e-01
59	1.75e+00	1.59e+00	9.07e-01	7.44e+05	4.57e+05	6.14e-01
67	2.55e+00	1.55e+00	6.10e-01	2.49e+06	9.53e+05	3.83e-01

Table 5.14 – Time for constructing proof trees and number of index accesses for all outputs for running CFL reachability analysis on PHP benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

Table 5.15 and Table 5.16 describe the time that is needed to populate the new indexes. The average time is 0.1 seconds, where benchmarks with longer times are also benchmarks that have comparably long runtime in the other phases. For all benchmarks, the index population time is negligibly small in comparison to the overall runtime.

The index population times for CFL reachability are shown in Table 5.17 and Table 5.18. The average time is 2 seconds, which also is negligible in comparison to the overall runtime.

Table 5.19 and Table 5.20 provide information on the overall memory overhead. Note that the memory usage for Soufflé's provenance is equivalent to what is reported in Table 5.3 and Table 5.4 as Soufflé's provenance does not require additional memory at this point. On average, the overall memory overhead is 51% for Java and 79% for PHP benchmarks. This includes the memory overhead that is due to the additional indexes as well as the memory overhead that is introduced at evaluation time due to the more complex instrumentation mentioned in the previous section.

The memory overhead for CFL reachability is shown in Table 5.21 and Table 5.22. On average, the overall memory overhead is 121% for Java and 187% for PHP benchmarks.

1	2.06e-02
2	1.39e-01
10	4.84e-03
17	2.24e-02
26	1.96e-03
29	9.87e-02
33	1.43e-02
38	2.34e-02
41	2.74e-03
43	2.82e-02
45	1.37e-02
46	1.10e+00
79	2.71e-02
80	5.82e-04
85	2.07e-02
103	6.34e-02

Table 5.15 – Time (s) for populating provenance indexes for running graph reachability analysis on Java benchmarks.

2	5.97e-02
7	1.03e-02
11	3.63e-02
12	3.58e-02
13	1.16e-02
14	1.03e-02
18	7.46e-02
26	4.72e-03
36	9.72e-02
44	4.50e-01
47	5.40e-01
56	2.31e-01
58	2.49e-01
59	7.86e-01
61	9.08e-02

Table 5.16 – Time (s) for populating provenance indexes for running graph reachability analysis on PHP benchmarks.

26	7.09e-02
39	4.23e+00
103	1.72e-01
110	4.65e+00
111	4.90e+00
112	3.90e+00

Table 5.17 – Time (s) for populating provenance indexes for running CFL reachability analysis on Java benchmarks.

0	4.42e+00
2	2.22e-01
7	3.54e-01
11	1.87e-01
12	1.91e-01
18	1.90e+00
30	3.63e+00
37	1.25e+00
42	3.43e+00
43	2.05e+00
47	6.69e-01
52	2.63e+00
56	1.65e+00
58	2.97e+00
59	5.83e-01
67	4.58e+00

Table 5.18 – Time (s) for populating provenance indexes for running CFL reachability analysis on PHP benchmarks.

	Maximum RSS (KB)		
	exp	sH	sH/exp
1	28968	35232	1.22e+00
2	115768	152172	1.31e+00
10	6452	8252	1.28e+00
17	12488	18704	1.50e+00
26	7720	8388	1.09e+00
29	29036	55236	1.90e+00
33	13688	17992	1.31e+00
38	20368	27472	1.35e+00
41	8132	9416	1.16e+00
43	18372	26696	1.45e+00
45	11860	16708	1.41e+00
46	157084	443544	2.82e+00
79	9308	16664	1.79e+00
80	5788	5736	9.91e-01
85	16240	22420	1.38e+00
103	15456	32936	2.13e+00

Table 5.19 – Maximum resident set size for running graph reachability analysis on Java benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

	Maximum RSS (KB)		
	exp	sH	sH/exp
2	11708	27364	2.34e+00
7	15784	18968	1.20e+00
11	13032	23372	1.79e+00
12	13212	23300	1.76e+00
13	10204	13796	1.35e+00
14	10928	13948	1.28e+00
18	53476	73540	1.38e+00
26	10612	12288	1.16e+00
36	110412	136596	1.24e+00
44	121852	228100	1.87e+00
47	67008	200144	2.99e+00
56	61076	119664	1.96e+00
58	71296	135644	1.90e+00
59	93816	295224	3.15e+00
61	54704	79716	1.46e+00

Table 5.20 – Maximum resident set size for running reachability analysis on PHP benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

	Maximum RSS (KB)		
	exp	sH	sH/exp
26	19260	34888	1.81e+00
39	623300	1431512	2.30e+00
103	32416	70632	2.18e+00
110	655232	1560324	2.38e+00
111	698484	1643440	2.35e+00
112	592412	1322468	2.23e+00

Table 5.21 – Maximum resident set size for running CFL reachability analysis on Java benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

	Maximum RSS (KB)		
	exp	sH	sH/exp
0	750296	1353816	1.80e+00
2	47228	84236	1.78e+00
7	69968	140956	2.01e+00
11	41648	77044	1.85e+00
12	41880	77196	1.84e+00
18	335112	664020	1.98e+00
30	652108	1204516	1.85e+00
37	214864	455196	2.12e+00
42	693404	1151452	1.66e+00
43	397600	685404	1.72e+00
47	129564	236276	1.82e+00
52	420308	867924	2.06e+00
56	285176	564812	1.98e+00
58	534308	972492	1.82e+00
59	115772	210296	1.82e+00
67	839888	1537344	1.83e+00

Table 5.22 – Maximum resident set size for running CFL reachability analysis on PHP benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

With respect to Hypothesis 3, it has to be noted that the memory overhead is significant. However, as can be seen in a comparison to the memory overheads during bottom up time, a big part of the overhead is due to the additional indexes. As outlined in Section 4.4, different implementation strategies might help to avoid the need to create separate indexes.

5.2.3 Magic Set Transformation

This section discusses how using Soufflé's magic set transformation influences the performance characteristics of subtree-heights provenance. Hypothesis 4 states that all other hypotheses also hold in presence of this transformation.

As explained in Section 3.1.1, Soufflé's magic set transformation modifies the rules before any other parts of the evaluation are done. In particular, magic set transformation is performed before the transformation step that adds the additional provenance attributes to the rules.

Furthermore, it is important to note that proof trees are generated using the rules that are the result of magic set transformation. Therefore, from the perspective of the provenance component applying a magic set transformation means computing provenance on a different Datalog program. From a user's perspective, who wants to compute proof trees for debugging

purposes, this fact might be problematic, as debugging the generated rules requires understanding how they relate to the original rules. Investigating the interactions between magic set transformations or other rule rewriting transformations with proof tree computations is an interesting direction which we leave as future work.

To understand the difference that magic set transformation makes for the performance of provenance computation, we reran the same experiments presented in the previous section for graph reachability and CFL reachability after applying magic set transformation to them.

In order to get a better understanding of the results for graph reachability, the modified rules are shown in Listing 5.7. It can be observed that new intermediary relations are generated, which are used in the body of rules of the original relations. In essence, while the tuples are filtered and less tuples are generated in every step of the evaluation, generating a single tuple might take more steps, which in the end results in proof trees with more nodes. The interested reader can find the modified rules for CFL reachability in Appendix A.2.1.

The detailed results for graph reachability can be found in Table A.3 to Table A.12 in the Appendix. The difference in performance between Soufflé's provenance and subtree-heights provenance is similar to the version without magic set transformation. The average memory overhead during bottom up computation is 25% for Java projects and 36% for PHP projects for subtree-heights compared to Soufflé's provenance. The average runtime overhead for bottom evaluation is 23%. The memory overhead during proof construction time is 52% for Java and 81% for PHP. The runtime for generating proof trees for all outputs is reduced by 73% Java and 77% for PHP projects. The number of index accesses is reduced by more than 90% in both cases.

For CFL reachabilty, the analysis with magic set transformation yields trees with significantly more levels and nodes than the unmodified analysis. As a consequence, for all considered benchmarks proof tree construction did not complete within the time limit of 15 minutes. In order to get an estimate of the performance, we therefore reran the experiments with a restriction on the height of the generated proof trees. Instead of generating full proof trees, we only computed the first 20 levels for every tree.

The detailed results that were obtained with this additional restriction can be found in Table A.13 to Table A.22 in the Appendix. The average memory overhead during bottom up computation is 26% for Java projects and 30% for PHP projects for subtree-heights compared to Soufflé's provenance. The average runtime overhead for bottom evaluation is 47% for Java projects and 36% for PHP projects. The memory overhead during proof construction time is 73% for Java and 86% for PHP. The runtime for generating proof trees for all outputs is reduced by 74% Java and 75% for PHP projects. The number of index accesses is reduced by more than 90% in both cases. Overall, those results differ considerably from the version without magic sets. In particular, the relative speed up for proof tree construction with subtree-heigths provenance is considerably bigger and the overheads are considerably smaller.

```
.decl +m0_path+_bf(x:symbol)
+m0_path+_bf(x) :-
   +m0_reachesSrcSink+_ff(),
   src(x).

+m0_path+_bf(x) :-
   +m0_path+_bf(x),
   src(x).

.decl +m0_reachesSrcSink+_ff()
+m0_reachesSrcSink+_ff().

.decl edge(x:symbol,y:symbol)
.output edge

.decl path+_bf(x:symbol,y:symbol)
path+_bf(x,y) :-
   +m0_path+_bf(x),
   edge(x,y).

path+_bf(x,y) :-
   +m0_path+_bf(x),
   src(x),
   path+_bf(x,z),
   edge(z,y).

.decl reachesSrcSink(x:symbol,y:symbol)
reachesSrcSink(arg0,arg1) :-
   reachesSrcSink+_ff(arg0,arg1).

.output reachesSrcSink

.decl reachesSrcSink+_ff(x:symbol,y:symbol)
reachesSrcSink+_ff(x,y) :-
   +m0_reachesSrcSink+_ff(),
   src(x),
   path+_bf(x,y),
   sink(y).

.decl sink(x:symbol)
.output sink

.decl src(x:symbol)
.output src
```

Listing 5.7 – Datalog rules for graph reachability after performing magic set transformation

The fact that for both analyses the observed speed ups were similar or bigger and the observed overheads were similar or smaller than in the versions without magic set transformation, provides evidence that indeed Hypothesis 1, 2, and 3 also hold in the presence of magic set transformations.

5.2.4 Summary

To sum up we draw the following conclusions for the considered hypotheses:

Hypothesis 1

Subtree-heights provenance reduces the number of index lookups at proof tree construction time and, hence, reduces the runtime of this phase.

For both analyses, the new provenance computation yielded a significant reduction in the number of index lookups. In particular, the average observed reduction was more than 90% for graph reachability and more than 50% for CFL reachability. For graph reachability, this resulted in an average observed speed-up of 80%. For CFL reachability, the average observed speed-ups were 6% for Java and 15% for PHP projects.

Hypothesis 2

Subtree-heights provenance runtime overhead on the bottom up time is not prohibitive.

The overhead in evaluation time was between 20% and 50% depending on the considered setup. While the overhead might be noticeable, it can be concluded that for the considered analyses it is not prohibitive.

Hypothesis 3

Subtree-heights provenance does not incur in dramatic memory overhead, neither during evaluation nor proof construction.

The memory overhead ranged from 50% to 180%. While this memory overhead is considerable, a big part of the overhead is due to the additional indexes used for proof tree construction. As outlined in Section 4.4, different implementation strategies might help to avoid the need to create separate indexes.

Hypothesis 4

The above hypotheses also hold with Soufflé's magic set transformation.

The experimental results showed that for both analyses the observed speed ups were similar or bigger and the observed overheads where similar or smaller in the versions with magic set transformation. This provides evidence that indeed Hypothesis 1, 2, and 3 also hold in the presence of magic set transformations.

6 Conclusion

We presented an alternative provenance evaluation strategy for the Soufflé Datalog engine that stores the height of the body tuples used in a tuple's generation in addition to Soufflé's rule and height annotations at evaluation time. Furthermore, we introduced a novel proof tree construction algorithm that uses the additional annotations to restrict the search space of potential body tuples and, hence, reduces the runtime of proof tree construction.

We formalized the new evaluation strategy by defining a new proof tree metric and showed its correctness. Moreover, we explained the modifications of the Soufflé engine that were needed for implementing the new provenance mechanism.

In the previous chapter, we presented the results of empirical experiments for two different Datalog program analyses run on mature real world Java and PHP projects. For both analyses, the new provenance computation yielded a significant speed-up in the proof tree computation time. In particular, for graph reachability the average observed speed-up was 80% and for CFL reachability the average observed speed-ups were 6% for Java and 15% for PHP projects. The overhead in evaluation time was between 20% and 50% depending on the considered setup. The memory overhead ranged from 50% to 180%. We showed that using Soufflé's magic set transformation results does not significantly change those results.

6.1 Future Work

There are several aspects of this project that can be extended in future work:

- A big portion of the memory overhead of the subtree-heights evaluation strategy is due to the additional indexes that are added especially for proof tree construction. As outlined in Section 4.4, one could explore different proof tree construction algorithms that do not rely on index lookups by the height of a tuple but on an additional filter operations after the index lookup.

- In the current implementation, tuples are annotated with the heights of the generating body tuples as well as the height of the tuple itself. As the overall height can be easily computed from the other height parameters, the memory overhead could be reduced by omitting to store the overall height. However, some modifications of the provenance index data structures will be needed in order to enable lookups by the overall height when it is not stored explicitly.
- At the moment, many parts of Soufflé's source code hard-code the used proof tree metric. A modification in the architecture could help to ease the experimentation with different proof tree metrics.
- Extending the experimental evaluation to different program analyses, for example analyses from the Doop [2] framework, might yield more insights on what characteristics of an analysis influence the effectiveness of using subtree-heights provenance.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[2] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. Porting Doop to Soufflé: A Tale of Inter-engine Portability for Datalog-based Analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, pages 25–30, New York, NY, USA, 2017. ACM.

[3] Tarun Arora, Raghu Ramakrishnan, William G. Roth, Praveen Seshadri, and Divesh Srivastava. Explaining program execution in deductive systems. In Stefano Ceri, Katsumi Tanaka, and Shalom Tsur, editors, *Deductive and Object-Oriented Databases*, pages 101–119, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[4] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pages 2:1–2:25, 2016.

[5] I. Balbin, G.S. Port, K. Ramamohanarao, and K. Meenakshi. Efficient bottom-up computation of queries on stratified databases. *The Journal of Logic Programming*, 11(3):295 – 344, 1991.

[6] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *The Journal of Logic Programming*, 10(3):255 – 299, 1991. Special Issue: Database Logic Progamming.

[7] Omar Benjelloun, Anish Das Sarma, Chris Hayworth, and Jennifer Widom. An introduction to ULDBs and the Trio system. *IEEE Data Eng. Bull.*, 29, 01 2006.

[8] Martin Bravenboer and Yannis Smaragdakis. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

[9] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *CoRR*, abs/1809.03981, 2018.

[10] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory — ICDT 2001*, pages 316–330, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[11] Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. Debugging of Wrong and Missing Answers for Datalog Programs with Constraint Handling Rules. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, PPDP '15, pages 55–66, New York, NY, USA, 2015. ACM.

[12] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989.

[13] James Cheney, Laura Chiticariu, and Wang-chiew Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1:379–474, 01 2009.

[14] Daniel Deutch, Amir Gilad, and Yuval Moskovitch. Selective Provenance for Datalog Programs Using Top-k Queries. *Proc. VLDB Endow.*, 8(12):1394–1405, August 2015.

[15] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, Dec 1999.

[16] B. Glavic and G. Alonso. Perm: Processing Provenance and Data on the Same Data Model through Query Rewriting. In *2009 IEEE 25th International Conference on Data Engineering*, pages 174–185, March 2009.

[17] Sergio Greco and Cristian Molinaro. Datalog and Logic Databases. *Synthesis Lectures on Data Management*, 10:1–169, 10 2015.

[18] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On Synthesis of Program Analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

[19] Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. A specialized B-tree for concurrent datalog evaluation. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, pages 327–339, 2019.

[20] Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis. Declarative datalog debugging for mere mortals. In Pablo Barceló and Reinhard Pichler, editors, *Datalog in Academia and Industry*, pages 111–122, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[21] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[22] Seokki Lee, Sven Köhler, Bertram Ludäscher, and Boris Glavic. Efficiently Computing Provenance Graphs for Queries with Negation. *CoRR*, abs/1701.05699, 2017.

[23] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. A User-guided Approach to Program Analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 462–473, New York, NY, USA, 2015. ACM.

[24] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. User-guided Program Reasoning Using Bayesian Inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 722–735, New York, NY, USA, 2018. ACM.

[25] Thomas W. Reps. Program Analysis via Graph Reachability. *Information & Software Technology*, 40:701–726, 1997.

[26] SonarSource SA. Sonar uCFG. https://github.com/SonarSource/sonar-ucfg, 2018.

[27] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On Fast Large-scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 196–206, New York, NY, USA, 2016. ACM.

[28] Pavle Subotic, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. Automatic Index Selection for Large-Scale Datalog Computation. *PVLDB*, 12(2):141–153, 2018.

[29] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical Virtual Method Call Resolution for Java. *SIGPLAN Not.*, 35(10):264–280, October 2000.

[30] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, pages 97–118, 2005.

[31] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. Effective Interactive Resolution of Static Analysis Alarms. *Proc. ACM Program. Lang.*, 1(OOPSLA):57:1–57:30, October 2017.

[32] David Zhao. Large-Scale Provenance for Soufflé . Master's thesis, The University of Sydney, Australia, 2017.

[33] David Zhao, Pavle Subotic, and Bernhard Scholz. Provenance for Large-scale Datalog. *CoRR*, abs/1907.05045, 2019.

A Appendix

A.1 Reachability Versions

Tables A.1 and A.2 show runtime and memory usage of bottom up evaluation for running four different configurations of the the graph reachability analysis presented in Section 5.1.2. For time as well as memory, the first column describes the analysis without modifications as presented in Listing 5.4. The second column contains results for the modified analysis presented in Listing 5.5 that considers only paths starting in source nodes. The third column describes the same analysis but with using Soufflé's magic set transformation, the fourth column again combines this analysis with magic set transformation.

It becomes apparent that there is no big performance difference between the last three configurations, while the first version of the analysis, i.e. the original version of the analysis without any magic set transformation, is several orders of magnitudes slower and uses several orders of magnitude more memory. For the first configuration, some benchmarks even do not stay within the maximum time of 15 minutes or the maximum memory of 20 GB, which leads to missing values in the table.

	Time (s)				Maximum RSS (KB)			
	standard	src	magic	src magic	standard	src	magic	src magic
1	nan	1.48e-01	1.35e-01	1.27e-01	nan	28784	27132	27296
2	nan	1.01e+00	1.04e+00	9.32e-01	nan	115628	111476	111296
10	4.74e+00	1.99e-02	2.01e-02	1.99e-02	112500	6164	6264	6288
17	1.41e+02	8.54e-02	8.46e-02	8.26e-02	2976292	12180	12108	12116
26	nan	1.61e-02	1.27e-02	1.32e-02	nan	7512	7304	7312
29	5.86e+02	4.44e-01	4.73e-01	4.46e-01	12091864	28696	27600	27864
33	nan	7.10e-02	6.80e-02	6.60e-02	nan	13412	13080	13076
38	7.59e+02	1.50e-01	1.32e-01	1.28e-01	15811080	20092	19116	19084
41	1.88e+01	1.64e-02	1.51e-02	1.68e-02	408312	7856	7760	7872
43	2.11e+02	1.31e-01	1.28e-01	1.26e-01	4612160	17984	17512	17740
45	1.21e+01	6.07e-02	5.78e-02	5.83e-02	289740	11728	11584	11460
46	nan	6.51e+00	6.73e+00	6.54e+00	nan	156888	154844	154896
79	1.11e+01	8.44e-02	8.40e-02	8.43e-02	254932	9080	8908	8792
80	2.55e+00	3.56e-03	3.94e-03	3.30e-03	64528	5680	5656	5732
85	2.92e+02	9.35e-02	8.69e-02	1.04e-01	6643088	16036	15376	15484
103	6.94e+01	2.36e-01	2.46e-01	2.41e-01	1616632	15144	15048	14900

Table A.1 – Bottom up time and maximum RSS for graph reachability without modifications (standard), with modification (src), without modification and with magic sets (magic) and with modifications and magic sets (src magic) on Java benchmarks.

	Time (s)				Maximum RSS (KB)			
	standard	src	magic	src magic	standard	src	magic	src magic
2	nan	2.08e-01	nan	2.10e-01	nan	11512	nan	11224
7	nan	7.37e-02	5.95e-02	5.89e-02	nan	15352	14212	14268
11	nan	1.37e-01	1.34e-01	1.32e-01	nan	12656	12276	12384
12	1.47e+01	1.34e-01	1.33e-01	1.31e-01	373840	12876	12284	12240
13	7.46e+00	5.37e-02	4.83e-02	5.62e-02	179292	9932	9716	9724
14	3.94e+01	5.34e-02	4.50e-02	4.50e-02	856636	10624	10260	10136
18	nan	5.03e-01	4.52e-01	4.74e-01	nan	53168	49832	49672
26	2.83e+01	3.42e-02	2.88e-02	2.74e-02	707612	10396	9628	9624
36	nan	8.43e-01	7.07e-01	7.26e-01	nan	110196	102216	102184
44	nan	2.74e+00	2.76e+00	2.81e+00	nan	121388	115404	115548
47	1.76e+02	2.40e+00	2.49e+00	2.52e+00	3919700	66428	66224	66144
56	nan	1.23e+00	1.33e+00	1.37e+00	nan	60868	58572	58716
58	1.88e+02	1.32e+00	1.26e+00	1.33e+00	4001200	71040	67808	67784
59	7.38e+01	3.51e+00	3.75e+00	3.78e+00	1777600	93544	94828	94680
61	6.09e+02	4.98e-01	4.89e-01	4.98e-01	13673808	54328	51136	51172

Table A.2 – Bottom up time and maximum RSS for graph reachability without modifications (standard), with modification (src), without modification and with magic sets (magic) and with modifications and magic sets (src magic) on PHP benchmarks.

A.2 Magic set transformation

A.2.1 CFL rules

The rules for CFL reachability that are generated by Soufflé's magic set transformation are displayed in Listing A.1.

```
// -- +m0_backward+_bf --
.decl +m0_backward+_bf(x:symbol)

+m0_backward+_bf(x) :-
   +m0_reachesSrcSink+_ff(),
   src(x).

+m0_backward+_bf(y) :-
   +m0_backward+_bf(x),
   path+_bf(x,y).

+m0_backward+_bf(y) :-
   +m0_backward+_bf(x),
   ret(x,y).



// -- +m0_forward+_bf --
.decl +m0_forward+_bf(x:symbol)

+m0_forward+_bf(y) :-
   +m0_reachesSrcSink+_ff(),
   src(x),
   backward+_bf(x,y).

+m0_forward+_bf(s) :-
   +m0_forward+_bf(s).

+m0_forward+_bf(s) :-
   +m0_forward+_bf(s).



// -- +m0_node+_b --
.decl +m0_node+_b(x:symbol)
```

```
36
37 +m0_node+_b(s) :-
38    +m0_backward+_bf(s).
39
40 +m0_node+_b(s) :-
41    +m0_forward+_bf(s).
42
43
44
45 // -- +m0_onestep+_bb --
46 .decl +m0_onestep+_bb(x:symbol,y:symbol)
47
48 +m0_onestep+_bb(x,y) :-
49    +m0_path+_bb(x,y).
50
51
52
53 // -- +m0_onestep+_bf --
54 .decl +m0_onestep+_bf(x:symbol)
55
56 +m0_onestep+_bf(x) :-
57    +m0_path+_bf(x).
58
59 +m0_onestep+_bf(x) :-
60    +m0_path+_bf(x).
61
62 +m0_onestep+_bf(x) :-
63    +m0_path+_bb(x,y).
64
65
66
67 // -- +m0_path+_bb --
68 .decl +m0_path+_bb(x:symbol,y:symbol)
69
70 +m0_path+_bb(entry,exit) :-
71    +m0_onestep+_bf(x),
72    continuation(x,y),
73    call(x,entry),
74    ret(exit,y).
75
76 +m0_path+_bb(z,y) :-
77    +m0_path+_bb(x,y),
```

```
78      onestep+_bf(x,z).
79
80  +m0_path+_bb(entry,exit) :-
81      +m0_onestep+_bb(x,y),
82      continuation(x,y),
83      call(x,entry),
84      ret(exit,y).
85
86
87
88  // -- +m0_path+_bf --
89  .decl +m0_path+_bf(x:symbol)
90
91  +m0_path+_bf(x) :-
92      +m0_backward+_bf(x).
93
94  +m0_path+_bf(x) :-
95      +m0_forward+_bf(s),
96      forward+_bf(s,x).
97
98  +m0_path+_bf(z) :-
99      +m0_path+_bf(x),
100     onestep+_bf(x,z).
101
102
103
104 // -- +m0_reachesSrcSink+_ff --
105 .decl +m0_reachesSrcSink+_ff()
106
107 +m0_reachesSrcSink+_ff().
108
109
110
111 // -- backward+_bf --
112 .decl backward+_bf(x:symbol,y:symbol)
113
114 backward+_bf(s,s) :-
115     +m0_backward+_bf(s),
116     node+_b(s).
117
118 backward+_bf(x,s) :-
119     +m0_backward+_bf(x),
```

```
120     path+_bf(x,y),
121     backward+_bf(y,s).
122
123 backward+_bf(x,s) :-
124     +m0_backward+_bf(x),
125     ret(x,y),
126     backward+_bf(y,s).
127
128
129
130 // -- call --
131 .decl call(x:symbol,y:symbol)
132
133 .output call
134
135
136
137 // -- continuation --
138 .decl continuation(x:symbol,y:symbol)
139
140 .output continuation
141
142
143
144 // -- edge --
145 .decl edge(x:symbol,y:symbol)
146
147 .output edge
148
149
150
151 // -- forward+_bf --
152 .decl forward+_bf(x:symbol,y:symbol)
153
154 forward+_bf(s,s) :-
155     +m0_forward+_bf(s),
156     node+_b(s).
157
158 forward+_bf(s,y) :-
159     +m0_forward+_bf(s),
160     forward+_bf(s,x),
161     path+_bf(x,y).
```

```
162
163 forward+_bf(s,y) :-
164    +m0_forward+_bf(s),
165    forward+_bf(s,x),
166    call(x,y).
167
168
169
170 // -- node+_b --
171 .decl node+_b(x:symbol)
172
173 node+_b(s) :-
174    +m0_node+_b(s),
175    edge(s,_).
176
177 node+_b(s) :-
178    +m0_node+_b(s),
179    edge(_,s).
180
181 node+_b(s) :-
182    +m0_node+_b(s),
183    call(s,_).
184
185 node+_b(s) :-
186    +m0_node+_b(s),
187    call(_,s).
188
189 node+_b(s) :-
190    +m0_node+_b(s),
191    ret(_,s).
192
193
194
195 // -- onestep+_bb --
196 .decl onestep+_bb(x:symbol,y:symbol)
197
198 onestep+_bb(x,y) :-
199    +m0_onestep+_bb(x,y),
200    edge(x,y).
201
202 onestep+_bb(x,y) :-
203    +m0_onestep+_bb(x,y),
```

```
204     continuation(x,y),
205     call(x,entry),
206     ret(exit,y),
207     path+_bb(entry,exit).
208
209
210
211 // -- onestep+_bf --
212 .decl onestep+_bf(x:symbol,y:symbol)
213
214 onestep+_bf(x,y) :-
215     +m0_onestep+_bf(x),
216     edge(x,y).
217
218 onestep+_bf(x,y) :-
219     +m0_onestep+_bf(x),
220     continuation(x,y),
221     call(x,entry),
222     ret(exit,y),
223     path+_bb(entry,exit).
224
225
226
227 // -- path+_bb --
228 .decl path+_bb(x:symbol,y:symbol)
229
230 path+_bb(x,y) :-
231     +m0_path+_bb(x,y),
232     onestep+_bb(x,y).
233
234 path+_bb(x,y) :-
235     +m0_path+_bb(x,y),
236     onestep+_bf(x,z),
237     path+_bb(z,y).
238
239 // -- path+_bf --
240 .decl path+_bf(x:symbol,y:symbol)
241
242 path+_bf(x,y) :-
243     +m0_path+_bf(x),
244     onestep+_bf(x,y).
245
```

```
246 path+_bf(x,y) :-
247    +m0_path+_bf(x),
248    onestep+_bf(x,z),
249    path+_bf(z,y).
250
251 // -- reachesSrcSink --
252 .decl reachesSrcSink(x:symbol,y:symbol)
253
254 reachesSrcSink(arg0,arg1) :-
255    reachesSrcSink+_ff(arg0,arg1).
256 .output reachesSrcSink
257
258 // -- reachesSrcSink+_ff --
259 .decl reachesSrcSink+_ff(x:symbol,y:symbol)
260
261 reachesSrcSink+_ff(x,z) :-
262    +m0_reachesSrcSink+_ff(),
263    src(x),
264    backward+_bf(x,y),
265    forward+_bf(y,z),
266    sink(z).
267
268 // -- ret --
269 .decl ret(x:symbol,y:symbol)
270 .output ret
271
272 // -- sink --
273 .decl sink(x:symbol)
274 .output sink
275
276 // -- src --
277 .decl src(x:symbol)
278 .output src
```

Listing A.1 – Datalog rules for CFL reachability after performing magic set transformation

A.2.2 Experimental results for graph reachability

Tables A.3 to A.12 contain the experimental results for the comparison between subtree-heights provenance and Soufflé's provenance on the graph reachability analysis.

	Bottom up time (s)				Bottom up maximum RSS (KB)			
	no	exp	sH	sH/exp	no	exp	sH	sH/exp
1	1.27e-01	1.16e-01	1.32e-01	1.13e+00	27296	27420	29076	1.06e+00
2	9.32e-01	8.39e-01	1.25e+00	1.48e+00	111296	111768	127668	1.14e+00
10	1.99e-02	1.94e-02	2.32e-02	1.19e+00	6288	6572	7032	1.07e+00
17	8.26e-02	8.21e-02	9.88e-02	1.20e+00	12116	12284	15524	1.26e+00
26	1.32e-02	1.21e-02	1.41e-02	1.17e+00	7312	7748	7556	9.75e-01
29	4.46e-01	4.31e-01	5.45e-01	1.26e+00	27864	28032	41712	1.49e+00
33	6.60e-02	7.02e-02	8.37e-02	1.19e+00	13076	13468	15412	1.14e+00
38	1.28e-01	1.38e-01	1.39e-01	1.01e+00	19084	19248	22148	1.15e+00
41	1.68e-02	1.41e-02	1.66e-02	1.18e+00	7872	8120	8552	1.05e+00
43	1.26e-01	1.33e-01	1.69e-01	1.27e+00	17740	17876	21588	1.21e+00
45	5.83e-02	7.09e-02	6.75e-02	9.52e-01	11460	11764	13892	1.18e+00
46	6.54e+00	6.48e+00	8.48e+00	1.31e+00	154896	155104	316092	2.04e+00
79	8.43e-02	8.50e-02	1.14e-01	1.34e+00	8792	9112	12956	1.42e+00
80	3.30e-03	3.18e-03	3.71e-03	1.17e+00	5732	5800	5752	9.92e-01
85	1.04e-01	8.57e-02	1.14e-01	1.34e+00	15484	15872	18844	1.19e+00
103	2.41e-01	2.34e-01	3.05e-01	1.30e+00	14900	15404	24700	1.60e+00

Table A.3 – Bottom up time and maximum RSS for running graph reachability analysis with magic sets on Java benchmarks without provenance (no), with Soufflé's explain provenance (exp), and with subtree-height provenance (sH).

	Bottom up time (s)				Bottom up maximum RSS (KB)			
	no	exp	sH	sH/exp	no	exp	sH	sH/exp
2	2.10e-01	2.12e-01	2.98e-01	1.41e+00	11224	11668	18368	1.57e+00
7	5.89e-02	6.17e-02	6.65e-02	1.08e+00	14268	14532	15220	1.05e+00
11	1.32e-01	1.35e-01	1.74e-01	1.29e+00	12384	12544	17068	1.36e+00
12	1.31e-01	1.43e-01	1.76e-01	1.24e+00	12240	12584	17132	1.36e+00
13	5.62e-02	5.09e-02	5.65e-02	1.11e+00	9724	9924	11108	1.12e+00
14	4.50e-02	4.61e-02	5.18e-02	1.12e+00	10136	10408	11664	1.12e+00
18	4.74e-01	4.43e-01	5.67e-01	1.28e+00	49672	50004	56300	1.13e+00
26	2.74e-02	2.87e-02	2.83e-02	9.84e-01	9624	9992	10352	1.04e+00
36	7.26e-01	7.32e-01	7.81e-01	1.07e+00	102184	102576	108572	1.06e+00
44	2.81e+00	2.80e+00	3.73e+00	1.33e+00	115548	115600	164580	1.42e+00
47	2.52e+00	2.52e+00	3.59e+00	1.42e+00	66144	66468	137116	2.06e+00
56	1.37e+00	1.37e+00	1.85e+00	1.35e+00	58716	59008	87784	1.49e+00
58	1.33e+00	1.36e+00	1.75e+00	1.29e+00	67784	67976	95872	1.41e+00
59	3.78e+00	3.72e+00	5.38e+00	1.44e+00	94680	94952	199700	2.10e+00
61	4.98e-01	5.16e-01	6.10e-01	1.18e+00	51172	51368	60184	1.17e+00

Table A.4 – Bottom up time and maximum RSS for running graph reachability analysis with magic sets on PHP benchmarks without provenance (no), with Soufflé's explain provenance (exp), and with subtree-height provenance (sH).

		exp		sH	
	# trees	# nodes	avg # nodes	# nodes	avg # nodes
1	1.79e+02	3.11e+04	1.74e+02	3.11e+04	1.74e+02
2	1.37e+03	2.37e+05	1.72e+02	2.37e+05	1.72e+02
10	1.62e+02	3.60e+04	2.22e+02	3.60e+04	2.22e+02
17	5.46e+02	1.40e+05	2.56e+02	1.40e+05	2.56e+02
26	6.20e+01	5.30e+03	8.55e+01	5.30e+03	8.55e+01
29	1.32e+03	2.97e+05	2.25e+02	2.97e+05	2.25e+02
33	8.40e+01	1.15e+04	1.36e+02	1.15e+04	1.36e+02
38	8.70e+01	1.13e+04	1.30e+02	1.13e+04	1.30e+02
41	7.40e+01	1.65e+04	2.23e+02	1.65e+04	2.23e+02
43	1.05e+02	2.30e+04	2.19e+02	2.30e+04	2.19e+02
45	6.80e+01	1.52e+04	2.24e+02	1.52e+04	2.24e+02
46	1.08e+04	1.39e+06	1.29e+02	1.39e+06	1.29e+02
79	2.50e+02	4.58e+04	1.83e+02	4.58e+04	1.83e+02
80	5.20e+01	1.20e+04	2.31e+02	1.20e+04	2.31e+02
85	5.60e+01	9.26e+03	1.65e+02	9.26e+03	1.65e+02
103	1.68e+03	1.52e+05	9.01e+01	1.52e+05	9.01e+01

Table A.5 – Number of trees and average number of nodes per tree for running graph reachability analysis with magic sets on Java benchmarks with Soufflé's explain provenance (exp) and with subtree-height provenance (sH).

		exp		sH	
	# trees	# nodes	avg # nodes	# nodes	avg # nodes
2	6.72e+02	1.07e+05	1.59e+02	1.07e+05	1.59e+02
7	4.92e+02	6.25e+04	1.27e+02	6.25e+04	1.27e+02
11	2.62e+03	2.76e+05	1.06e+02	2.76e+05	1.06e+02
12	2.62e+03	2.76e+05	1.06e+02	2.76e+05	1.06e+02
13	9.67e+02	1.43e+05	1.48e+02	1.43e+05	1.48e+02
14	1.16e+03	2.33e+05	2.02e+02	2.33e+05	2.02e+02
18	2.71e+03	2.70e+05	9.97e+01	2.70e+05	9.97e+01
26	2.23e+02	1.70e+04	7.61e+01	1.70e+04	7.61e+01
36	7.10e+01	5.97e+03	8.41e+01	5.97e+03	8.41e+01
44	3.81e+03	5.60e+05	1.47e+02	5.60e+05	1.47e+02
47	8.83e+03	1.19e+06	1.34e+02	1.19e+06	1.34e+02
56	1.25e+04	1.51e+06	1.20e+02	1.51e+06	1.20e+02
58	1.57e+04	2.84e+06	1.81e+02	2.84e+06	1.81e+02
59	6.99e+04	6.71e+06	9.59e+01	6.71e+06	9.59e+01
61	1.20e+03	1.42e+05	1.18e+02	1.42e+05	1.18e+02

Table A.6 – Number of trees and average number of nodes per tree for running graph reachability analysis with magic sets on PHP benchmarks with Soufflé's explain provenance (exp) and with subtree-height provenance (sH).

	Proof tree construction time (s)			Number of index lookups		
	exp	sH	sH/exp	exp	sH	sH/exp
1	1.13e+01	1.61e+00	1.42e-01	7.36e+07	4.06e+06	5.51e-02
2	3.58e+02	4.97e+01	1.39e-01	1.82e+09	9.48e+07	5.22e-02
10	6.44e-01	2.16e-01	3.36e-01	3.65e+06	1.79e+05	4.91e-02
17	7.68e+00	1.52e+00	1.97e-01	4.66e+07	3.15e+06	6.77e-02
26	2.91e-02	2.71e-02	9.34e-01	3.74e+04	9.18e+03	2.45e-01
29	5.26e+01	7.81e+00	1.48e-01	3.20e+08	1.89e+07	5.91e-02
33	1.48e+00	2.72e-01	1.84e-01	9.92e+06	7.45e+05	7.52e-02
38	3.29e+00	4.81e-01	1.46e-01	2.27e+07	1.56e+06	6.85e-02
41	4.01e-01	1.23e-01	3.06e-01	2.36e+06	1.24e+05	5.26e-02
43	3.31e+00	5.00e-01	1.51e-01	2.46e+07	1.09e+06	4.42e-02
45	2.60e-01	9.71e-02	3.74e-01	1.38e+06	7.85e+04	5.70e-02
46	6.85e+02	1.33e+02	1.94e-01	3.67e+09	3.31e+08	9.02e-02
79	1.19e+00	3.46e-01	2.92e-01	6.60e+06	4.35e+05	6.60e-02
80	1.92e-01	7.21e-02	3.76e-01	1.33e+06	6.31e+04	4.76e-02
85	1.62e+00	2.39e-01	1.47e-01	1.21e+07	6.34e+05	5.23e-02
103	1.01e+01	2.38e+00	2.34e-01	6.45e+07	6.47e+06	1.00e-01

Table A.7 – Time for constructing proof trees and number of index accesses for all outputs for running graph reachability analysis with magic sets on Java benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

	Proof tree construction time (s)			Number of index lookups		
	exp	sH	sH/exp	exp	sH	sH/exp
2	2.33e+00	7.31e-01	3.14e-01	1.45e+07	9.07e+05	6.23e-02
7	3.32e+00	8.23e-01	2.48e-01	1.75e+07	1.94e+06	1.11e-01
11	9.29e+00	2.58e+00	2.78e-01	5.16e+07	5.07e+06	9.83e-02
12	9.21e+00	2.59e+00	2.81e-01	5.16e+07	5.07e+06	9.83e-02
13	3.46e+00	1.10e+00	3.17e-01	1.77e+07	1.43e+06	8.04e-02
14	1.12e+01	2.24e+00	2.00e-01	7.10e+07	4.05e+06	5.71e-02
18	1.18e+02	2.41e+01	2.03e-01	4.86e+08	5.61e+07	1.15e-01
26	1.14e+00	2.64e-01	2.31e-01	6.85e+06	7.18e+05	1.05e-01
36	3.02e+00	5.18e-01	1.71e-01	1.40e+07	1.33e+06	9.47e-02
44	5.04e+02	6.35e+01	1.26e-01	1.97e+09	1.22e+08	6.19e-02
47	1.24e+02	2.49e+01	2.01e-01	8.63e+08	7.14e+07	8.28e-02
56	3.87e+02	8.71e+01	2.25e-01	1.65e+09	2.01e+08	1.22e-01
58	1.14e+02	2.55e+01	2.24e-01	6.40e+08	3.77e+07	5.89e-02
59	2.59e+02	7.06e+01	2.72e-01	1.56e+09	1.67e+08	1.07e-01
61	2.94e+01	6.37e+00	2.17e-01	1.89e+08	1.71e+07	9.05e-02

Table A.8 – Time for constructing proof trees and number of index accesses for all outputs for running graph reachability analysis with magic sets on PHP benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

1	1.03e-02
2	1.05e-01
10	4.09e-03
17	1.79e-02
26	5.74e-04
29	8.43e-02
33	1.13e-02
38	1.60e-02
41	1.58e-03
43	2.38e-02
45	1.16e-02
46	1.07e+00
79	2.48e-02
80	3.04e-04
85	1.61e-02
103	5.47e-02

Table A.9 – Time (s) for populating provenance indexes for running graph reachability analysis with magic sets on Java benchmarks.

2	5.91e-02
7	3.36e-03
11	3.18e-02
12	3.13e-02
13	8.29e-03
14	6.87e-03
18	4.57e-02
26	1.22e-03
36	4.18e-02
44	4.03e-01
47	5.27e-01
56	2.11e-01
58	2.11e-01
59	8.26e-01
61	6.23e-02

Table A.10 – Time (s) for populating provenance indexes for running graph reachability analysis with magic sets on PHP benchmarks.

A.2.3 Experimental results for CFL reachability

Tables A.13 to A.22 contain the experimental results for the comparison between subtree-heights provenance and Soufflé's provenance on the CFL reachability analysis.

	Maximum RSS (KB)		
	exp	sH	sH/exp
1	27420	30780	1.12e+00
2	111768	142628	1.28e+00
10	6572	8328	1.27e+00
17	12284	18540	1.51e+00
26	7748	8000	1.03e+00
29	28032	55884	1.99e+00
33	13468	17380	1.29e+00
38	19248	25112	1.30e+00
41	8120	8816	1.09e+00
43	17876	25728	1.44e+00
45	11764	16236	1.38e+00
46	155104	483380	3.12e+00
79	9112	17412	1.91e+00
80	5800	5752	9.92e-01
85	15872	21604	1.36e+00
103	15404	34700	2.25e+00

Table A.11 – Maximum resident set size for running graph reachability analysis with magic sets on Java benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

	Maximum RSS (KB)		
	exp	sH	sH/exp
2	11668	28896	2.48e+00
7	14532	15748	1.08e+00
11	12544	22792	1.82e+00
12	12584	22852	1.82e+00
13	9924	12752	1.28e+00
14	10408	12780	1.23e+00
18	50004	63812	1.28e+00
26	9992	10676	1.07e+00
36	102576	115872	1.13e+00
44	115600	222780	1.93e+00
47	66468	219136	3.30e+00
56	59008	120104	2.04e+00
58	67976	132280	1.95e+00
59	94952	326864	3.44e+00
61	51368	70656	1.38e+00

Table A.12 – Maximum resident set size for running graph reachability analysis with magic sets on PHP benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

	Bottom up time (s)				Bottom up maximum RSS (KB)			
	no	exp	sH	sH/exp	no	exp	sH	sH/exp
26	6.83e+00	6.84e+00	1.05e+01	1.53e+00	102156	102400	115032	1.12e+00
39	6.29e-01	6.30e-01	9.06e-01	1.44e+00	11024	11308	13816	1.22e+00
103	3.89e+01	3.84e+01	5.81e+01	1.51e+00	110196	110464	147084	1.33e+00
110	4.85e+01	4.81e+01	7.24e+01	1.50e+00	124420	124576	169708	1.36e+00
111	4.33e+01	4.31e+01	6.46e+01	1.50e+00	110484	110668	150920	1.36e+00

Table A.13 – Bottom up time and maximum RSS for running CFL reachability analysis with magic sets on Java benchmarks without provenance (no), with Soufflé's explain provenance (exp), and with subtree-heights provenance (sH).

	Bottom up time (s)				Bottom up maximum RSS (KB)			
	no	exp	sH	sH/exp	no	exp	sH	sH/exp
2	1.23e+01	1.24e+01	1.82e+01	1.47e+00	24864	24952	43288	1.73e+00
7	4.05e-01	4.14e-01	5.16e-01	1.25e+00	17072	17472	19480	1.11e+00
11	3.69e-01	3.72e-01	4.91e-01	1.32e+00	11196	11548	13480	1.17e+00
12	3.79e-01	3.68e-01	5.17e-01	1.40e+00	11396	11608	13296	1.15e+00
18	1.75e+00	1.70e+00	2.18e+00	1.28e+00	51812	52044	55944	1.07e+00
30	8.71e+00	8.42e+00	1.22e+01	1.45e+00	76676	76832	89784	1.17e+00
37	1.16e+01	1.15e+01	1.60e+01	1.39e+00	59920	60272	84196	1.40e+00
47	6.28e+01	6.23e+01	9.53e+01	1.53e+00	67172	67220	122692	1.83e+00
56	5.96e-01	5.89e-01	6.81e-01	1.16e+00	39920	40040	41912	1.05e+00

Table A.14 – Bottom up time and maximum RSS for running CFL reachability analysis with magic sets on PHP benchmarks without provenance (no), with Soufflé's explain provenance (exp), and with subtree-heights provenance (sH).

		exp		sH	
	# trees	# nodes	avg # nodes	# nodes	avg # nodes
26	62	4.88e+05	7.88e+03	2.15e+05	3.47e+03
39	148	2.29e+06	1.55e+04	1.97e+06	1.33e+04
103	385	3.41e+06	8.85e+03	2.01e+06	5.22e+03
110	271	1.93e+06	7.13e+03	9.84e+05	3.63e+03
111	303	2.06e+06	6.79e+03	1.13e+06	3.72e+03

Table A.15 – Number of trees and average number of nodes per tree for running CFL reachability analysis with magic sets on Java benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

		exp		sH	
	# trees	# nodes	avg # nodes	# nodes	avg # nodes
2	5.50e+01	6.99e+05	1.27e+04	6.25e+05	1.14e+04
7	5.80e+01	4.00e+05	6.89e+03	3.89e+05	6.71e+03
11	7.20e+01	5.19e+05	7.20e+03	1.60e+05	2.22e+03
12	7.20e+01	5.19e+05	7.20e+03	1.60e+05	2.22e+03
18	2.92e+02	3.15e+06	1.08e+04	2.79e+06	9.55e+03
30	2.83e+02	3.35e+06	1.18e+04	1.98e+06	6.98e+03
37	5.61e+02	5.04e+06	8.98e+03	4.09e+06	7.29e+03
47	1.86e+02	6.97e+05	3.75e+03	2.36e+05	1.27e+03
56	1.33e+02	1.31e+06	9.84e+03	4.12e+05	3.10e+03

Table A.16 – Number of trees and average number of nodes per tree for running CFL reachability analysis with magic sets on PHP benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

	Proof tree construction time (s)			Number of index lookups		
	exp	sH	sH/exp	exp	sH	sH/exp
26	2.40e+01	4.16e+00	1.73e-01	101555248	5498352	5.41e-02
39	6.98e+01	2.72e+01	3.89e-01	309651039	17608636	5.69e-02
103	6.20e+02	1.16e+02	1.86e-01	-1255719903	221401164	-1.76e-01
110	4.15e+02	5.99e+01	1.44e-01	2006422092	102422760	5.10e-02
111	3.85e+02	6.09e+01	1.58e-01	1829130860	103248477	5.64e-02

Table A.17 – Time for constructing proof trees and number of index accesses for all outputs for running CFL reachability analysis with magic sets on Java benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

	Proof tree construction time (s)			Number of index lookups		
	exp	sH	sH/exp	exp	sH	sH/exp
2	2.60e+01	9.13e+00	3.52e-01	1.36e+08	9.28e+06	6.82e-02
7	8.87e+00	5.18e+00	5.84e-01	3.09e+07	2.75e+06	8.92e-02
11	1.86e+01	2.61e+00	1.41e-01	1.14e+08	3.41e+06	3.00e-02
12	1.87e+01	2.60e+00	1.39e-01	1.14e+08	3.41e+06	3.00e-02
18	8.71e+01	3.51e+01	4.03e-01	3.59e+08	1.19e+07	3.31e-02
30	2.01e+02	3.88e+01	1.93e-01	1.00e+09	4.16e+07	4.14e-02
37	6.71e+02	1.30e+02	1.94e-01	3.73e+09	2.05e+08	5.49e-02
47	1.23e+02	8.47e+00	6.90e-02	7.85e+08	1.65e+07	2.10e-02
56	3.79e+01	5.99e+00	1.58e-01	2.09e+08	5.16e+06	2.47e-02

Table A.18 – Time for constructing proof trees and number of index accesses for all outputs for running CFL reachability analysis with magic sets on PHP benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

benchmark	1.03e-02
26	1.97e-01
39	2.10e-02
103	3.76e-01
110	4.61e-01
111	4.06e-01

Table A.19 – Time (s) for populating provenance indexes for running CFL reachability analysis with magic sets on Java benchmarks.

2	1.79e-01
7	2.94e-02
11	1.91e-02
12	2.19e-02
18	7.74e-02
30	1.91e-01
37	2.49e-01
47	5.58e-01
56	4.66e-02

Table A.20 – Time (s) for populating provenance indexes for running CFL reachability analysis with magic sets on PHP benchmarks.

	Maximum RSS (KB)		
	exp	sH	sH/exp
26	102400	148068	1.45e+00
39	11308	17752	1.57e+00
103	110464	211740	1.92e+00
110	124576	248092	1.99e+00
111	110668	221176	2.00e+00

Table A.21 – Maximum resident set size for running CFL reachability analysis with magic sets on Java benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).

	Maximum RSS (KB)		
	exp	sH	sH/exp
2	24952	74680	2.99e+00
7	17472	24792	1.42e+00
11	11548	16828	1.46e+00
12	11608	16756	1.44e+00
18	52044	68268	1.31e+00
30	76832	121228	1.58e+00
37	60272	126812	2.10e+00
47	67220	215384	3.20e+00
56	40040	49544	1.24e+00

Table A.22 – Maximum resident set size for running CFL reachability analysis with magic sets on PHP benchmarks with Soufflé's explain provenance (exp) and with subtree-heights provenance (sH).