View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by PolyPublie

UNIVERSITÉ DE MONTRÉAL

CHANGE-BASED APPROACHES FOR STATIC TAINT ANALYSES

NICOLAS CLOUTIER
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2018

© Nicolas Cloutier, 2018.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

CHANGE-BASED APPROACHES FOR STATIC TAINT ANALYSES

présenté par: CLOUTIER Nicolas
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. GAGNON Michel, Ph. D., président
M. MERLO Ettore, Ph. D., membre et directeur de recherche
M. FOKAEFS Marios-Eleftherios, Ph. D., membre

DEDICATION

To all my family and friends who supported me
in this ridiculous adventure.

ACKNOWLEDGEMENTS

I want to thank my supervisor, Ettore Merlo, for supporting me during my two years under his supervision. The knowledge he shared and the time he spent guiding and revising my work is invaluable.

I would like to thank my supervisor at IBM Advanced Studies, John Peyton, for his incredible support of the project. It is your help that made my success possible.

I want to thank IBM and the people at IBM Centre for Advanced Studies for giving me this opportunity. I especially want to thank Vio Onut for his support in this project.

To my colleagues at Polytechnique – your ideas and suggestions improved my work. You made this thesis possible.

I would also like to thank the Natural Sciences and Engineering Research Council for sponsoring me.

Finally, I would like to thank Fonds de recherche du Québec – Nature et technologies for their economic support.

ABSTRACT

In the past few years, many security problems have been discovered in all kinds of software. For some of these vulnerabilities, ill-intentioned people exploited them and successfully stole information about people and companies. The monetary cost of these vulnerabilities is real, and for this reason, many developers are trying to find these vulnerabilities before hackers do.

One method to find vulnerabilities in an application before it is published is to use a static analysis tool. Taint analysis is one static analysis technique that is closely related to security. This approach simulates how data is propagated inside the application with the goal of finding locations that could either leak sensitive information or damage the integrity of the system.

These analyses can take hours, depending on the tool used and the size of the codebase being analyzed. On top of taking considerable time, these analyses tend to be repeated over and over during development. The computation of taint is exhaustive and usually redone from scratch on every execution, even if the codebase has stayed nearly the same since the last analysis. This is why our work will mainly focus on how we can take advantage of the incremental nature of software development to accelerate the computation of taint.

We propose new techniques to update the taint information from the changes in the source code between two versions of a given software. Our approaches are granular to the lines of code and succeed at greatly reducing the time required to find potential vulnerabilities in the projects that we analyzed.

RÉSUMÉ

Les logiciels développés dans les dernières années ont souvent été aux prises avec des vulnérabilités qui ont été exploitées par des personnes mal intentionnées. Certaines de ces attaques ont coûté cher à plusieurs entreprises et particuliers dus aux données volées. Ainsi, il y a un besoin réel de déceler ces failles dans le code avant leur utilisation.

Une technique pour tenter de détecter de possibles vulnérabilités dans le code avant même que le logiciel soit public est d'utiliser un outil d'analyse statique. En utilisant plus particulièrement l'analyse de teinte qui a pour but de simuler la propagation de données critiques dans le programme, il est possible de trouver des points d'accès dans le code qui ne sont pas protégés.

Cependant, ces analyses peuvent prendre des heures selon l'outil utilisé et le volume de code à analyser. En plus d'être de longues analyses, elles sont souvent effectuées à répétition sur le même code au fur et à mesure qu'il est développé. Chaque fois, c'est un calcul exhaustif à partir de zéro alors que les changements dans le code sont généralement minimes en comparaison au volume total du logiciel. Instinctivement, on pourrait supposer que si les changements sont mineurs dans le code, les changements dans les résultats devraient aussi l'être. Dans ce mémoire, nous nous penchons sur des techniques tirant avantage de la nature incrémentale du développement logiciel afin d'accélérer le calcul de la teinte.

Notre travail propose une technique novatrice qui met à jour la teinte en fonction des changements dans le code entre deux versions. Nous utilisons une technique qui est granulaire à la ligne de code. Avec nos améliorations, nous réussissons à largement réduire le temps de calcul nécessaire sur les projets que nous avons analysés.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

ABSTRACT v

RÉSUMÉ vi

TABLE OF CONTENTS vii

LIST OF TABLES x

LIST OF FIGURES xi

LIST OF SYMBOLS AND ABBREVIATIONS xii

CHAPTER 1 INTRODUCTION 1
- 1.1 Motivation for Change-based Taint Analyses 1
- 1.2 Thesis Statement 2
- 1.3 Theoretical Background 2
  - 1.3.1 Control Flow Graph 2
  - 1.3.2 Static Single Assignment 3
  - 1.3.3 Data Flow Graph 3
  - 1.3.4 Taint 3
  - 1.3.5 Data-Flow Algorithms 3
  - 1.3.6 Trace Generation 4
  - 1.3.7 Source Control Management 4
- 1.4 Problematic Elements 5
- 1.5 Research Objectives 6
- 1.6 Thesis organization 6

CHAPTER 2 CRITICAL LITERATURE REVIEW 7
- 2.1 Taint Analysis for Java 7
- 2.2 Andromeda 7
- 2.3 Cheetah 8
- 2.4 Path Verification 9

2.5 FlowDroid 9
2.6 Reviser 9

CHAPTER 3 ARTICLE 1 : CHANGE-BASED APPROACHES ON TAINT ANALYSES 11
3.1 Introduction 11
3.2 Incremental Taint Analysis 13
3.2.1 Main concepts of taint analysis 13
3.2.2 Equivalent classes of sinks 17
3.2.3 Implementation of standard taint analysis 19
3.2.4 Inter-procedural strategy 19
3.2.5 Road to an incremental approach 22
3.2.6 Research Questions: 28
3.3 Differential Trace Generation 29
3.3.1 Trace Generation 29
3.3.2 Differential Strategy 30
3.3.3 Research Questions 32
3.4 Experimental Design 32
3.5 Results 34
3.5.1 Taint analysis 34
3.5.2 Trace generation 37
3.5.3 Generic 39
3.6 Threats to validity 41
3.7 Related work 42
3.8 Conclusion 43
3.9 Acknowledgements 44

CHAPTER 4 TAINT REACHABILITY OPTIMIZATION 45
4.1 Introduction 45
4.2 Approach 45
4.3 Research Questions 46
4.4 Methods 46
4.5 Results and Discussion 48
4.6 Threats to Validity 48
4.7 Conclusion 50

CHAPTER 5 GENERAL DISCUSSION 51

5.1 Discussion of Research Objectives 51
5.2 Integration in the Developer Workflow 52
5.2.1 Integration System 52
5.2.2 Interactive Developer Environment 53

CHAPTER 6 CONCLUSION AND RECOMMENDATIONS 55
6.1 Summary of Contributions 55
6.2 Limitations of Proposed Solution 56
6.3 Future Research 56

BIBLIOGRAPHY 58

LIST OF TABLES

Table 3.1 Metrics on the inter-procedural context strategy 39

LIST OF FIGURES

Figure 3.1 Example of SSA instructions forming a control-flow graph 15
Figure 3.2 Example of a data-flow graph 15
Figure 3.3 Example of a converged graph after an execution 23
Figure 3.4 Example of a converged graph after the removal of nodes and edges . 26
Figure 3.5 Example of a converged graph after the addition of a node and an edge 28
Figure 3.6 Impact on incremental analyses in relation to flow changes in WebGoat 35
Figure 3.7 Impact on incremental analyses in relation to flow changes in Roller . 35
Figure 3.8 The ratio of nodes processed in the incremental taint analysis and a global taint analysis 36
Figure 3.9 The performance of incremental taint analysis and global analysis as it relates to data-flow changes 37
Figure 3.10 Distribution of traces generated by commit 38
Figure 3.11 Proportion of commits with and without new differential traces . . . 38
Figure 3.12 Performance of differential trace generation as it relates to data-flow changes 40
Figure 3.13 Comparison of mean time required for a regular analysis compared to our change-based approaches 42
Figure 4.1 Histogram of nodes traversed for shortest path traces generated with and without the taint reachability optimization on WebGoat 5.4 . . . 49
Figure 4.2 Histogram of time for shortest-path traces generated with and without the taint reachability optimization on WebGoat 5.4 49
Figure 5.1 Example of the integration of the taint analysis tool Cheetah [14] in a IDE. 53

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
AST	Abstract Syntax Tree
CFG	Control Flow Graph
DFG	Data Flow Graph
FRQNT	Fonds de recherche du Québec Nature et technologies
IBM	International Business Machines Corporation
IDE	Inter-procedural Distributive Environment
IDFG	Inter-procedural Data Flow Graph
IFDS	Inter-procedural, Finite, Distributive Subset
JDK	Java Development Kit
NSERC	Natural Sciences and Engineering Research Council
OWASP	Open Web Application Security Project
RAM	Random-Access Memory
SCM	Source Control Management
SQL	Structured Query Language
SSA	Static Single Assignment
TAJ	Taint Analysis for Java
URL	Uniform Resource Locator
WALA	T.J. Watson Libraries for Analysis
XSS	Cross-Site Scripting

CHAPTER 1 INTRODUCTION

1.1 Motivation for Change-based Taint Analyses

The work of software developers is frequently at the mercy of ill-intentioned individuals. Hackers try to gain access to systems by all available means because the potential gains are so interesting. A breach in the database of a popular website may easily compromise the private information of millions of users and cause severe damage for both the users and the website. Passwords can be sold to third parties, and stolen credit cards and social security numbers may be used to usurp identities. Web applications, in particular, are sensitive to these attacks and must be designed with care.

Many of these attacks are using injection flaws, which were cited as the most critical security risk in both the 2013 and 2017 reviews of the Open Web Application Security Project (OWASP) Foundation [5]. This attack consists of injecting untrusted data into a query, like a Structured Query Language (SQL) environment, with the goal of executing an unintended and unauthorized behavior. There are static analyses that exist to try to automatically detect these vulnerabilities in the source code. Taint analysis being one of them that simulates how tainted, or distrusted, data is propagated in the software. Developers do not use these analyses as much as they should, since they are framed as slow and imprecise [13]. This is where we want to change things. By reducing the time of a taint analysis, we can remove one of the hindrances of development. A faster taint analyzer could be used in more projects and be performed more frequently by users. We suppose that a more frequent use of such a tool will help detect vulnerabilities sooner and reduce their lifespan, consequently helping to make web applications more secure.

We could play with precision parameters to accelerate an analysis while trying to minimize impacts on the results. However, there is another avenue that could be much more effective: the change-based approach. The evolution of a software project is generally iterative over its course of development. In the lifespan of a project, the source code will evolve in multiple small increments. With each increment, the majority of the codebase remains unchanged. This may not be true for every project and every iteration, but we suspect it is generalized enough to use it to our advantage.

Our proposed solution uses these changes to drive a taint analyzer and update the taint only when needed. It will, as much as possible, utilize the previous results. If the changes in the source code are small, the changes in the results of the analysis should also be small and

reflect changes in the source code.

We will detect the changes concerning security in an application with respect to what can be found with a taint analyzer. This will then be used to report new vulnerabilities to the developers on top of the existing vulnerabilities. A full analysis will report all the vulnerabilities detected. However, without an extra step to memorize which of the vulnerabilities are new or old, a developer may miss a new vulnerability that has just been added. By using a change-based approach, we can detect very easily which of these are new issues and report them in a way that clearly classifies them as new.

Our motivations for this thesis are: (1) to help developers find the security impacts of changes in source code and (2) to reduce the lifespan of security policy violations in the source code by reducing the computing time of taint analysis.

1.2 Thesis Statement

Prior research [11] has already studied change-based approaches for static analyses, but the taint problem has not been specifically tested; we believe we can provide an alternative that is easier to understand and, possibly, faster.

Our thesis is as follows:

> Driven by changes in source code, we can perform more efficient and faster taint analyses by reducing the parts of the graph explored and, subsequently, use it to find impacted vulnerabilities.

1.3 Theoretical Background

1.3.1 Control Flow Graph

Our security analysis, like other data-flow analyses, must follow the flow of the data inside the application. These analyses generally use a Control Flow Graph (CFG) [10]. This graph $CFG = \langle V, E \rangle$ has instructions as vertices V, and the edges E consist of all the possible execution paths between the instructions. Such a graph is built from the Abstract Syntax Tree (AST), a data structure representing the overall application and, more importantly, its hierarchical components. It is by exploring the tree and, more precisely, the instructions that we find the control flow edges in the CFG and build it. However, our framework is already giving us the CFG, and we do not have to bother with this part of the analyzer.

1.3.2 Static Single Assignment

In our project, the framework building the control flow graph operates on Java bytecode, and for this reason, instructions use the Static Single Assignment (SSA) form [10]. This alternative format transforms the usual instructions found in the source code into another format to ensure that every variable may not be assigned by more than one instruction. Among other things, this creates a new variable for each assignment, and it adds new instructions to merge the new variables together, when needed, to ensure the same behavior as dictated by the original instructions. Such a form is useful in static analyses because there is only one valid definition for a variable and not multiple ones where we don't know if they are reachable or not for a given instruction.

1.3.3 Data Flow Graph

The representation used in our project is different from the CFG. With this last graph, we present another one that specifically represents the data flow. This Data Flow Graph (DFG), $DFG = \langle V, E \rangle$, has, for vertices V, the parameters of every SSA instruction, including the returning value of an instruction. The edges E represent the possible propagations of the data or, in other terms, the links between the definitions and the usages of a variable. For an example, see Figure 3.1 and Figure 3.2 in Section 3.2.1.

1.3.4 Taint

A tainted variable is a variable that may contain unsafe data for an application. In the context of web applications, unsafe data refers to data coming from an external source, like user input or the network, that is not properly sanitized and cannot be trusted.

The goal of our taint analysis is to find such variables in the source code and to detect the vulnerabilities Application Programming Interface (API) with tainted parameters.

1.3.5 Data-Flow Algorithms

Such data-flow algorithms rely on the concept of fixed point. The taint values computed in the graph will grow monotonically until the algorithms converge and reach a fixed point [10]. This occurs by imposing a partial order, or a semi-lattice, to the computed values. In the case of a taint analysis, the variables will generally start untainted, but the algorithm will gradually make them tainted as it explores the graph. For more details about the exact data-flow algorithms and equations, refer to Section 3.2.1.

1.3.6 Trace Generation

We define the term trace as an example of the exploitation of a vulnerability. It is usually represented as an executable path from one instruction to another. In the context of taint, a path would be the ordered set of instructions executed to carry unsafe data from a user input to a vulnerable API. In our experiments, traces are generated for every vulnerability found in the taint analysis.

1.3.7 Source Control Management

Since the algorithms are executed on multiple iterations of the same project, we use a Source Control Management (SCM) tool to fetch these revisions. We chose git [2] since it is a popular tool, and many open-source projects are freely available on this platform. The revisions, or commits, are grouped in pairs to be analyzed in our change-based analysis. The differences between the pairs are generated, and it is these differences that will drive our algorithms.

1.4 Problematic Elements

The main problematic element is the computation time for taint analysis. We analyzed relatively small projects of thousands of lines of code, and our analysis took around 10 minutes for a traditional analysis used as a baseline. This is relatively similar to other research projects [27]. Even if an execution of mere minutes is not bothersome for small projects, industrial projects can have a codebase many times larger than that. For industrial projects, the analysis will not scale well and could take hours to complete, depending on the precision required.

It is important to improve performance, even for small projects. An ideal analyzer should be fast enough to enable interactive feedback inside an integrated development environment. If we succeed at increasing the performance to bring this interactivity, it could help to popularize security tools in the developer workflow. Many developers are not using such tools exactly because of the slow speed of the analyses [13].

Algorithms need to have a sufficient level of precision to bring relevant results. This is crucial in handling the interprocedural propagations since we would miss too many vulnerabilities without it. The strategy chosen to manage this will greatly impact precision and performance [20].

In addition, modern languages have additional mechanisms, which can make our algorithms more complex. The fields of a class need additional logic to precisely propagate the taint. This is similar to precisely propagating a taint over an element of a collection or a pointer modified with pointer arithmetic. The use of statics, exceptions, and many other semantics will all have an effect on the precision of a taint analysis.

In an ideal situation, we want to maintain a somewhat similar level of precision while reducing the time required to compute an analysis. To do such a thing, our solution is to implement a change-based approach. We must use old results as much as possible, find the differences, and limit our computations to the differences and their impacts. The differences must be found in the source code and reflected in the data-flow graphs to compute the taint changes.

In a similar fashion, a trace must show an example of every vulnerability found in the taint analysis. The complex elements of taint analysis are also applicable to the generation of traces. It is inefficient to generate the traces of every vulnerability at each analysis and even more so when there is little change in the source code since the last execution. The traces will mainly be the same, and we could simply use the old traces when applicable.

1.5 Research Objectives

Our main research objective is to reduce the computation time of taint analysis. We will implement a change-based strategy to incrementally update the taint values. Our analysis will be driven by the changes in the source code and will limit the scope to only recompute the impacted parts of the DFG.

Our second research objective is to reduce the computation time of the traces. We will implement a change-based strategy to find new traces and limit the generation to only them. Our analysis will be driven by the changes found in the taint analysis.

1.6 Thesis organization

Chapter 2 presents a literature review of various research projects related to taint analysis and incremental strategies used in static analysis. Chapter 3 presents our main article, which describes our innovative, incremental taint analysis algorithms; the experimentations done; and the results obtained. Chapter 4 describes an additional experiment on trace generation that was not included in the article. Chapter 5 is a general discussion of the various subjects of the article. Finally, Chapter 6 is a conclusion summarizing our contributions, our main limitations, and future research.

CHAPTER 2 CRITICAL LITERATURE REVIEW

Multiple projects have worked on improving the speed and reactiveness of static analysis. In this chapter, we present some of these works that are closely related to taint analysis. This chapter focus on the various experimentations that are related to change-based strategies.

2.1 Taint Analysis for Java

The work of Tripp et al. [27] on Taint Analysis for Java (TAJ) was one of the first stepping stones for our current research. They use T.J. Watson Libraries for Analysis (WALA) [9] as a static analysis framework and extended their improvements on top of it. This research has a lot of similarities to the current study because we are both analyzing Java web applications, and we have both collaborated with International Business Machines Corporation (IBM) inside the software IBM AppScan Source [4].

This work is primarily known for the creation of a priority-driven call-graph construction. With this, they can avoid a full computation of their static analyses and give partial results to the user. Their algorithm includes a budget of time and of memory, and under these conditions, the call graph is monotonically generated until the limits are reached. Because of this, the approach under-approximates, which may miss some results. A direct consequence is that their taint analysis is not conservative, but they suggest that the impact on precision is a compromise worth taking.

2.2 Andromeda

The next project, Andromeda [28], is again mainly related to the construction of the call graph. However, Andromeda, which is a direct successor of TAJ [28], is more interesting than TAJ for us. The authors still want to increase the scalability of taint analysis and have improved some of the issues associated with TAJ. Their main contribution is an on-demand approach for their analysis. Important data structures, like the call graph, are no longer eagerly constructed, but a lazier strategy is used instead. Also, the alias analysis is only performed on fields that are tainted. In this new version, their analyses are now sound and conservative.

The next part of their work, which is closely related to ours, is the implementation of incremental taint analysis. Their strategy is to recompute all the methods that may have been changed or impacted. One of our main critiques is the lack of detail in which they present

their strategy. It is not clear what they have done, but the granularity of their incremental algorithm seems to be limited to functions. They keep the results of a function when it is not impacted by the changes and recomputes the others. They do not analyze pairs of versions with many changes, only those with small, incremental changes. They add or delete a statement or a method and test the response time. Compared to them, we present a replicable approach and measure against code revisions freely available on internet.

2.3 Cheetah

A more recent work is the project Cheetah [14]. While previous projects mainly focused on the scalability of data structures and analyses adjacent to taint analysis, this project is focused on making the computation of the taint itself faster.

They use a priority-driven strategy or, as they call it, Just-In-Time. Usually, data flow analysis starts from the entry points of an application and will compute its flow functions over every statement that is reachable [10]. Cheetah's approach is somewhat different. Cheetah will also analyze the methods that are unreachable from the entrypoints of the application. This is one of the main points of their strategy, their scanner will starts from the method on which the developer is currently focused and not from an entry point. It is an interactive approach intended to be used while the developer is working on its code.

This is possible because their tool is integrated as an Eclipse plugin. Starting with this method, they can propagate the taint over the statements based on a specific priority. They use a priority queue to order the statements to be processed for computing the taint until a fixed point is reached. Locality is the main factor of importance of a statement for the priority queue. Using this method, a statement of the currently focused function is much more important than a statement accessible over a virtual call. With this, they can find taint on the vulnerable sinks close to the developer's current point of focus almost instantaneously and report them to the user inside the developer environment.

They have the same goal as us, which is to give to the developer the taint results sooner. Yet, their approach is totally different from our change-based strategy. They have done an integration with Eclipse, and one of their next objectives is to implement an incremental analysis while we have done the opposite.

2.4 Path Verification

On the other hand, Le and Pattison [18] are doing something closer to what we want to accomplish. Their main work is done on the CFG, which is annotated and transformed to contain vertices and edges of multiple versions at the same time. This results in what they call a multiversion interprocedural control flow graph. This is a useful data structure, which they mainly use to verify if a patch has been correctly applied to the version changes.

They have developed an incremental analysis for their system, which is able to detect bugs, like buffer overflow and null pointer dereference. For finding the bugs, they use their special CFG, which is incrementally updated from the previous version to a new one corresponding to the changes. They use the cached results of previous analyses, and with them, they query the analyses on the new parts of the graph. However, this procedure is not really developed or explained. They lack algorithms to explain the specific incremental strategy, and they lack performance results to show the improvements over a regular analysis. We present an approach that is more detailed, and we will focus more on the time reduction.

2.5 FlowDroid

One of the most powerful open-source taint analysis engines at the moment is FlowDroid [12]. By integrating some of the previous strategies, like the on-demand approach of Andromeda, they obtained great results and they are scalable for bigger projects. It is a static analysis engine optimized for Android applications and Inter-procedural, Finite, Distributive Subset (IFDS) problems. In fact, multiple recent projects implement IFDS solvers, like WALA [9]. These types of solvers are powerful because they are able to solve classic data flow problems, like reaching definitions, live variables, and other Gen/Kill algorithms, in a polynomial time while being able to handle interprocedural analyses [23].

2.6 Reviser

The authors of FlowDroid tried to improve the performances of the IFDS solver itself. Arzt and Bodden [11] worked on Reviser to make their solver fully incremental for IFDS and Inter-procedural Distributive Environment (IDE) problems. IDE is simply an extension of IFDS that solves more problems and supports more languages [26]. Reviser is an interesting project because it is close to our objective and approach.

They developed an algorithm that uses the changes in the CFG to update the computed values of the data flow analysis impacted by the changes found in the source code [11]. They

have found whether or not the nodes of the CFG are safe, i.e. if it does not have a reachable predecessor that has changed. They will purge the results of the unsafe nodes and repropagate the flow functions on the changed nodes and their impacted neighbors. Finally, they require a second phase where they will again iterate all changed merge points, which are statements with more than one predecessor, to ensure the propagation of unchanged predecessors and ensure valid results.

Reviser is for general IFDS problems and not specialized to taint, so we are different since our experimentations are specific to taint. Also, we differ on the strategy used for the incremental analysis itself. The difference lies in how we update the flow values or, in our case, the taint. They clear the values and compute them again to ensure monotonicity while we replace the clearing by a careful update. This operation will be explained in detail in the following chapters.

CHAPTER 3 ARTICLE 1 : CHANGE-BASED APPROACHES ON TAINT ANALYSES

Nicolas Cloutier, Ettore Merlo and John Peyton
Submitted to the Journal of Systems and software

Abstract

Modern web applications are sensitive to multiple types of vulnerabilities that put millions of users at risk of having their information compromised. Even if an application's vulnerabilities were all eliminated at some point, there would still be a risk of introducing new vulnerabilities when it is updated. Code review and other manual techniques, as well as automated methods, such as static analysis, can be used to find new defects. However, since performance is an issue, they are usually not performed on every commit but executed daily by an integration server or manually by a security expert.

This is why we propose novel change-based approaches to perform faster inter-procedural taint analyses in the context of software development. An evaluation of timings on the open-source projects WebGoat and Roller shows performance gains of 90 to 95% on our analyzed projects compared to the analysis without the incremental improvements. This dramatic reduction in the required computation time would enable developers to quickly analyze code changes and detect potentially threatening vulnerabilities.

Keywords

inter-procedural static analysis, taint analysis, incremental analysis

3.1 Introduction

Web applications are similar to other software programs, such as mobile applications, video games, and embedded systems, and may be the target of multiple attacks since they contain sensitive data desired by hackers. They are at risk due to their exposure to public networks and need security policies to ensure the confidentiality, integrity, and availability of data [22]. A policy is a formal or informal description of the processes and mechanisms used to partition and secure a system. Vulnerabilities are violations of these policies, which can be dangerous, especially when they are exploited, and lead to unexpected behaviors. Briefly, vulnerabilities can be classified into two categories: application vulnerabilities and environmental vulner-

abilities [6]. Application vulnerabilities arise from the program itself, while environmental vulnerabilities are violations caused by something external to the application, for example, the absence of segregation on an internal network for critical systems.

Application vulnerabilities are generally caused by either faulty code or design flaws [6]. SQL injection, Cross-Site Scripting (XSS), and unintended authentication are known examples [5] of this type of violation. We are specifically interested in data-driven vulnerabilities where external data, which may be malicious, is propagated to critical parts of the system.

Data-driven vulnerabilities are interesting since static analysis techniques like taint analysis can preemptively detect some of these [28]. Taint analysis is used to detect violations of policies by finding paths between external sources and vulnerable calls to API. To find these paths, we generate a justification in the form of a trace, presented to the developer. These traces have the goal of explaining how the vulnerability can be exploited by showing an example of how it is propagated in the application. We focus our research on improving the taint analyses and the traces generation.

Performing taint analyses may require a significant amount of time on large systems due to the volume of code to analyse and traditionally do not scale well [27]. It is possible to perform the analyses more quickly by trading precision for better performance [27], but it could deter developers from using these tools [13].

Another way to make them faster while avoiding a trade-off is using change-based approaches [11, 19, 24]. Change-based approaches are strategies that detect changes between pairs of versions and reflect changes in the analyses to avoid computation of unchanged results. It is a more generic term and includes, for example, incremental and differential analyses.

A differential analysis will detect changes, discards the analysis results affected by the changes, and then repeats the analysis to re-computes the missing analysis results. An incremental analysis, however, does not discard the results; it updates them. The first method is limited to finding differences, while the latter goes one step further and uses the differences to incrementally perform a computation. Other approaches, such as on-demand and partial analysis [28], control and limit when and where a computation occurs. They are not necessarily change-based, but it is possible to combine them [14].

As already reported by multiple researchers [11, 19, 24], one of the main opportunities to improve data-flow analyses is considering the incremental nature of software development. Programs evolve with multiple small iterations, and we expect them to remain the same with some small modifications after each new revision, but exceptions may exist for major revisions or major refactoring. Therefore, we should expect the results of a program's static

analyses to be quite similar and the differences to reflect changes made in the code. We will account for the evolutionary nature of software development and present new and original change-based approaches to improve inter-procedural data-flow approaches for taint analysis. By reducing the amount of time required to perform computations that analyse the impact of code changes, we hope to be more time efficient and improve the algorithms' scalability for industrial software.

Our incremental and differential algorithms perform comparisons of pairs of software versions. Commonly, a pair of versions includes two commits coming from source control where one version is the oldest and the other is the newest. Our analyses store the static analysis results of versions and use it as a starting point for the next analysis. When a change-based analysis is triggered, changes in the flow graph are computed from differences in the source code, and their impact on taint and other analyses results are conservatively computed.

To present our change-based approaches, the paper is structured as follows. Section 3.2 introduces and provides a detailed description of our novel strategy to incrementally update a taint analysis. Section 3.3 explains how we can directly use the results of an incremental taint analysis to accelerate trace generation. Section 3.4 explains the experiment's methodology, while Section 3.5 presents and discusses the results. Threats to validity are discussed inSection 3.6, and related works are discussed in Section 3.7. Finally, conclusions are presented in Section 3.8.

3.2 Incremental Taint Analysis

3.2.1 Main concepts of taint analysis

In this subsection, we define the main concepts related to taint analysis. A reader may skip these concepts and go directly to the next subsection if desired.

Our system builds a control-flow graph $CFG = \langle V_{CFG}, E_{CFG} \rangle$ where the vertexes V_{CFG} are static single assignment (SSA) instructions, and the edges E_{CFG} represents how flow control can be transferred between SSA nodes. Figure 3.1 is an example of possible SSA instructions.

We also create another representation as a data-flow graph $DFG = \langle V_{DFG}, E_{DFG} \rangle$ of the SSA arguments. In this graph, the vertexes V_{DFG} represent each parameter of an SSA instruction. The directional edges E_{DFG} represent how data flow from one parameter to another. For every use of a defined node, an edge is created. When we use the terms successor and predecessor, we usually refer to the vertexes connected by an edge to a vertex of the DFG. A predecessor of a node is another node in the DFG which can be the origin of the data. A successor of a node is another node in the DFG where the data can be propagated into. The dashed lines

and nodes handle the inter-procedural aspects, which are detailed in Section 3.2.3. For the CFG in Figure 3.1, the corresponding DFG is Figure 3.2.

In the context of this research, the meaning of 'taint' is close to the concept of contamination. We define a tainted variable as a variable that can contain external and untrusted data that has not been properly sanitized. Such a variable could be corrupted by a malicious user to ultimately violate established security policies. Our analyses operate on DFG, and the taint will be computed for every node in V_{DFG}.

A taint source defines a function, or an API, which returns untrusted data. Usually, it is any function that returns values coming from a user or the network. We mark the return value from a source as tainted. Subsequently, the variables that the return value can flow into will also be tainted. Validators (or downgraders [25]) enforce security policies. They are special functions that "downgrade" the level of taint. In the case of SQL injection, they will sanitize the input by, for example, escaping specific characters, such as apostrophes, and make it safe to use for SQL queries. We represent the taint in function of sources and validators formally in Equation 3.1.

For any node n in V_{DFG}, it is tainted if a path p exists in the DFG from a source n_0 and this node n. n_0 is another node of the DFG, which is also in the set of *sources*. To be tainted, there should not be any node in the path n_x, which is in the sets of *validators*. Such a path bringing taint from a source to a node may also be called an unprotected path. A protected path means that the taint from the source is blocked by a validator. If all the paths from the sources to a node are protected by validators, we cannot mark the node as tainted.

$$Taint(n) = \begin{cases} True & \exists p_{\langle n_0,\ldots,n\rangle} \in DFG \text{ such that} \\ & \quad n_0 \in sources \wedge \nexists n_x \in p \mid n_x \in validators \\ False & \textbf{otherwise} \end{cases} \tag{3.1}$$

However, if a node executing a SQL query is tainted, it has a possible vulnerability. We call such vulnerable APIs a security sink. As indicated in Eq. 3.2, a vulnerability exists only if a node n of the DFG is tainted and is a sink or in a set of *sinks*.

$$Vulnerable(n) = \begin{cases} True & Taint(n) \wedge n \in sinks \\ False & \textbf{otherwise} \end{cases} \tag{3.2}$$

Our goal is to compute the taint in all nodes of the data-flow graph, and to detect tainted sinks, we need data-flow analyses. In the literature [10], a standard data-flow analysis frame-

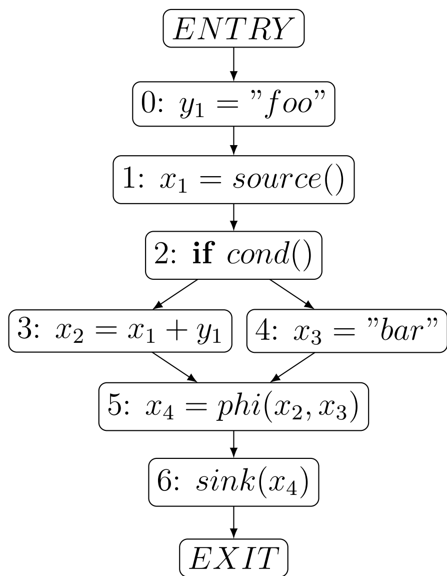

Figure 3.1 Example of SSA instructions forming a control-flow graph

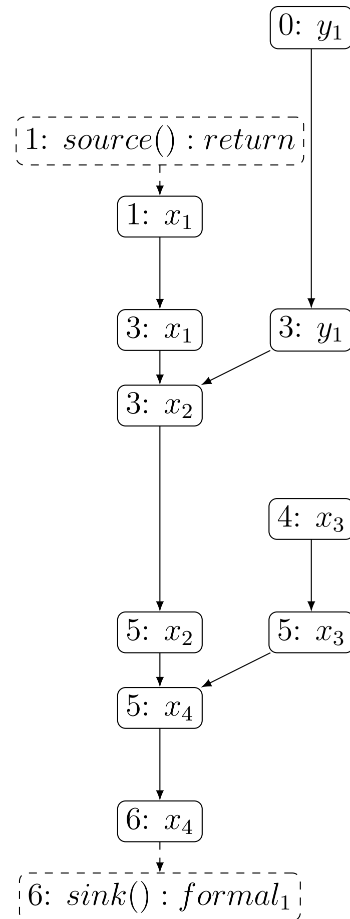

Figure 3.2 Example of a data-flow graph

work has been defined to work on flow graphs of basic blocks, and derived from this format, we can define an iterative algorithm to compute the taint over the SSA instructions with Algorithm 1. The main parameters for this forward algorithm are the domain of values V_{DFG}, flow equations f_D, and meet operator $\vee$.

Algorithm 1 Iterative algorithm for a forward data-flow taint problem

1: **for** each vertex D from the DFG V_{DFG} **do**
2: $OUT[D] = \bot$
3: **while** changes to any OUT occur **do**
4: **for** each vertex D from from the DFG V_{DFG} **do**
5: $IN[D] = \bigvee_{P \text{ a predecessor of } D} OUT[P]$
6: $OUT[D] = f_D(IN[D])$

The dictionary OUT contains the taint processed for every vertex of V_{DFG}, and IN is somewhat similar. It corresponds to the inputted taint coming from the predecessors of a vertex.

The values computed over the domain V_{DFG} are represented by a simple boolean lattice. It does denote whether the variable related to a vertex is tainted [15, 27]. The Boolean value $False$ means untainted, and $True$ means tainted.

In the algorithm, we chose to iteratively propagate the taint from the sources to the sinks. It is a forward approach. As previously defined in Equation 3.1, a node is tainted if a path exists between a source and node without a validator on the path. For our algorithm, this means that the meet operator $\vee$ must be a logical disjunction $\vee$. If a predecessor of a node is tainted, the value contained in IN for this node must also be tainted, as represented in Equation 3.3.

$$In(D) = \begin{cases} True & \textbf{if } \vee_{p \in predecessors(D)} Out(p) \textbf{ is true} \\ False & \textbf{otherwise} \end{cases} \tag{3.3}$$

We will define the flow equations f_D as follows in Equation 3.4. It is a modified version of GEN & KILL functions as seen in the literature [10], but it is adapted to Boolean logic. It does receive the Boolean taint value x for a node D as an input and computes the taint. It does use two functions, $Gen(D)$ and $Kill(D)$, which are defined in the following equations.

$$f_D(x) = Gen(D) \vee (x \wedge \neg Kill(D)) \tag{3.4}$$

The $GEN(D)$ function (Equation 3.5) will generate new taint if the current node is a source.

To do so, it needs a set *sources* of the existing sources. The kill function (Equation 3.6) will stop the propagation of taint if the current node is a validator. If a node is in the set of validators *validators*, the returned value for the KILL function is true so that the flow equation outputs a taint value equal to false. Subsequently, the function $f_D(x)$ will return the taint for a node as true if either the node is a source or the inputted value is tainted and not killed by a validator.

$$Gen(D) = \begin{cases} False & \text{if } D \notin sources \\ True & \text{if } D \in sources \end{cases} \tag{3.5}$$

$$Kill(D) = \begin{cases} False & \text{if } D \notin validators \\ True & \text{if } D \in validators \end{cases} \tag{3.6}$$

Since we are generating new taint and propagating it over the graph, the starting values must be equal to $\perp$ or not-tainted. By iteratively marking nodes as tainted and never unmarking a node, we ensure a monotonic algorithm and eventually converge to a fixed point.

3.2.2 Equivalent classes of sinks

Since validators are used to enforce only a subset of security policies, they will only protect a subset of the sinks. A validator for SQL injections will transform strings to make them harmless for an SQL query function, a sink. However, for another sink, it may not perform a sufficient protection. Consequently, the taint is different depending on the sink analyzed. The previous equation 3.1 of taint should be modified to include this detail in Equation 3.7. It is modified to associate the taint with sinks and only downgrades when a validator related to the sinks exists in the path. It is done by having $validators(s)$, which return the set of validators that can protect a sink s. The other variables are identical to those defined in Equation 3.1.

$$SinkSensitiveTaint(n, s) = \begin{cases} True & \exists p_{\langle n_0, \ldots, n \rangle} \in DFG \text{ such that} \\ & \quad n_0 \in sources \wedge \nexists n_x \in p \mid n_x \in validators(s) \\ False & \textbf{otherwise} \end{cases} \tag{3.7}$$

As taint must be considered differently for every sink (i.e., API) analyzed, Eq. 3.7 should be computed for every distinct type of API.

Repeating the computation of taint can become time-consuming. To give an idea of the number of analyses required, on the version 5.4 of WebGoat, an open-source project which we will talk more about in Section 3.4, there are 42 different API sinks and 37 validators. So taint analysis should be repeated 42 times for the whole application.

By investigating different APIs, we found that most of them seem to behave in a similar manner in terms of taint. For example, there are multiple sinks related to SQL, and they interact identically with all the validators. For these sinks, the set of validators that downgrade the taint is equal. In this case, it is possible to construct equivalent classes and group them together to reduce taint analysis computation effort. If the number of equivalent classes is low, it will drastically reduce the computations since a taint analysis is required for every class.

As presented in the equations (3.8), we provide the following definitions. A validator v is in the validators $Vals_s$ of a sink s only if this API validator blocks the taint for this API sink. Similarly, a sink s is also in the set of sinks $Sinks_v$ of a validator v when the taint is blocked for this sink and validator combination.

A sink s is in an equivalent class of sinks S_e if and only if they are protected by exactly the same set of API validators.

A validator v is in an equivalent class of validators V_e if and only if they protect exactly the same set of API sinks.

$$\begin{aligned} v \in Validators_s &\iff doesValidate(v, s) = True \\ s \in Sinks_v &\iff doesValidate(v, s) = True \\ s \in S_e &\iff \forall s' \in S_e(Validators_{s'} = Validators_s) \\ v \in V_e &\iff \forall v' \in V_e(Sinks_{v'} = Sinks_v) \end{aligned} \tag{3.8}$$

For the projects analyzed in this paper, we computed that the number of equivalent sinks and validators was much smaller than the number of APIs. By performing a brute-force comparison of every sink and validator, we obtained a maximum of five distinct groups of equivalent sinks and three groups of equivalent validators. For example all the sinks and validators for SQL injections are now grouped toghether.

These groups are used in our algorithm by simply replacing the sink for Equation 3.7 with the corresponding class of equivalence. Consequently, the kill equation (3.6) is modified to use validators corresponding to the current equivalency class of sinks.

On our test samples, we use five computations to find the same set of tainted sinks as 42

analyses without these classes. On this reduced computation analysis, we observed the same tainted sinks at a fraction of the computation time normally required. We passed from 72 minutes to 9 minutes for our analysis.

3.2.3 Implementation of standard taint analysis

Based on the previously defined equations for taint, we designed the Algorithm 2 and implemented it using the application IBM Security AppScan Source [4]. In this algorithm, the successors function takes as a parameter an element of V_{DFG} and returns a subset of the V_{DFG} where $successors(n) = \{s|(n, s) \in E_{DFG}\}$. Similarly, the predecessors function has the same domain and image, but the subset returned corresponds to $predecessors(n) = \{p|(p, n) \in E_{DFG}\}$. E_{DFG} is given by our framework and enables us to avoid the management of complex mechanisms, such as fields.

An Inter-procedural Data Flow Graph (IDFG) is an alternative version of the data-flow graph used for our inter-procedural analyses and is discussed below in Section 3.2.4.

According to Algorithm 1 defined in Section 3.2.1, we iterate every vertex from the DFG repeatedly until a fixed point is reached. This approach is inefficient and can be improved. A better strategy would be to find the analyzed sources in the project and begin our computation from this set. To detect the sources, we must start from the entry points in the application, typically the main function, and recursively search in the code for the source API. This improvement is reflected in lines 5-7 and the Algorithm 3. The function $FetchAllInterprocContexts(source)$ used take a source and retrieve all the corresponding nodes in the inter-procedural data-flow graph (IDFG).

In Algorithm 1, we iterate V_{DFG} repeatedly until a fixed point is reached. It is possible for our problem to compute the same results with a work list. It is valid since the only nodes impacted by a change of taint are the transitive successors of the node impacted. By queuing all the successors when a change occurs, we ensure the correct result is eventually achieved. We expect such a modification to accelerate our computation.

3.2.4 Inter-procedural strategy

Obviously, this new algorithm needs to be inter-procedural, since programming languages have procedures that determine how data can be propagated.

One approach for a fully precise analysis would clone, or inline, the graph of each function for every possible call stack [10]. In practice, this approach is not feasible in a reasonable amount of time for programs with a relevant size [20]. However, an insensitive strategy would not

Algorithm 2 Global fixed-point taint analysis

1: **procedure** GLOBALANALYSIS($idfg, entrypoints$)
2:     **for all** $n \in idfg$ **do** ▷ Initialization of taint for every node
3:         $taint[n] \leftarrow False$
4:     $worklist \leftarrow List()$ ▷ Push the sources to launch the analysis
5:     **for all** $source \in ReachableSources(entrypoints)$ **do**
6:         **for all** $node \in FetchAllInterprocContexts(source)$ **do**
7:             $worklist.push(node)$
8:     **while** $\neg worklist.empty()$ **do**
9:         $c \leftarrow worklist.pop()$
10:         **if** $c \in sources$ **then**
11:             $inTaint \leftarrow True$
12:         **else if** $c \in validators$ **then**
13:             $inTaint \leftarrow False$
14:         **else**
15:             $inTaint \leftarrow False$
16:             **for all** $p \in predecessors(c)$ **do**
17:                 $inTaint \leftarrow inTaint \vee taint[p]$
18:         **if** $inTaint \wedge \neg taint[c]$ **then** ▷ Inside the if the node is tainted
19:             $taint[c] \leftarrow inTaint$
20:             **if** $c \in sinks$ **then**
21:                 **Report tainted sink**
22:             **for all** $s \in successors(c)$ **do**
23:                 $worklist.push(s)$

Algorithm 3 Search of reachable sources

```
1:  function ReachableSources(entrypoints)
2:      reachableSources ← Set()
3:      visited ← Set()
4:      q ← Queue()
5:      for all entrypoint ∈ entrypoints do
6:          q.enqueue(entrypoint)
7:      while ¬q.empty() do
8:          function ← q.dequeue()
9:          for all node ∈ dfg(function) do
10:             if node ∈ sources then
11:                 reachableSources ← reachableSources ∪ node
12:             if isCallSite(node) then
13:                 for all calledFunction ∈ called(node) do
14:                     if calledFunction ∉ visited then
15:                         visited.insert(function)
16:                         q.enqueue(calledFunction)
        return reachableSources
```

propagate values over the inter-procedural edges and only resolve what is inside a procedure. We have chosen to compute the taint of a function and the related inter-procedural edges separately for every existing call site. This strategy offers a balance between speed and the precision of results for our analysis.

Let the inter-procedural data-flow graph $IDFG = (V, E)$ be an exploded version of the data-flow graphs (DFG). The IDFG is a graph containing every clone created for inter-procedurality. A clone will be created for a function for every existing context, which is the call-site of the caller. For every context, an inter-procedural edge is created from the parameter node of the calling function to the formal parameters of the called functions. An edge will also be created from the return value of the called function to the actual return value at the call-site.

One important aspect of modern languages affecting our algorithms is the use of polymorphisms. Virtual calls can be analyzed to narrow the set of possible called functions. Our work environment is already providing us with a virtual call resolution, and the results are used directly by our analyses.

By using the context-sensitive strategy of creating a clone for every caller-callee pair possible, the size of the IDFG should be multiple times larger. Furthermore, the time required to compute the taint should also be considerably longer. We must also remember that a separate

analysis must be performed for each equivalent class of sinks present in an analyzed project.

However, we have an algorithm that can be used as a baseline in our experiments against our incremental algorithms.

For a single run of our analysis, a node can be processed multiple times by the main while loop (Line 8) of Algorithm 2. First, source nodes will be added to the work list to begin the computation. Second, a node computed in the work list can push its successors in the work list only when the taint increases, which can only happen once and is enforced by the condition at line 18 of the algorithm. In the worst case, all of the nodes in the IDFG will be reachable from the sources and will be treated by the algorithm.

3.2.5 Road to an incremental approach

An incremental taint analysis takes the flow values from previous analyses and uses them as a starting point. In this section, we will describe how our algorithms implement this idea.

Incrementally updating a taint analysis will bring new complexities, such as handling multiple versions of source code and the need for new algorithms capable of handling the differences. We want to reduce the number of nodes processed to only the nodes impacted by changes, but we must ensure our new algorithms will find the same set of tainted sinks compared to our baseline analysis. Even with these constraints, we want to investigate whether, like similar projects that had good results [11], an incremental analysis can be faster than our baseline analysis on an average pair of versions.

Furthermore, we have a differential tool that works on a pair of versions and computes the changes in the IDFG between the pair. Algorithm 4 is a generic approach used to compute various sets between the pair. This generic approach is expensive but can be improved, for example, if the developer environment is keeping track of the changes as they occur.

Our incremental algorithm is based on the strategy of juxtaposing the impacts of every change. In this section, we explain how we juxtapose the effects of removals and additions in the graphs to update the taint.

Consider an example on Figure 3.3 with a taint already computed on it. In this graph, the node B is tainted, thus its reachable successors, D, E, and F, are all tainted as well since there are no validators to block the propagation. It will be used later as a small example of incremental analysis.

Considering the equation of taint presented in Equation 3.1, the taint is the existence of an unprotected path from a source to a node. For any node in the graph, the removal of another node or an edge will only affect the taint value if it is in a tainted path between

Algorithm 4 Diff algorithm

```
1: function ComputeChanges(idfg_old, idfg_new)
2:     ⟨N_old, E_old⟩ ← idfg_old
3:     ⟨N_new, E_new⟩ ← idfg_new
4:     nodes+ ← ∅
5:     nodes− ← ∅
6:     edges+ ← ∅
7:     edges− ← ∅
8:     for all n ∈ N_old do
9:         if n ∉ N_new then
10:            nodes− ← nodes− ∪ n
11:    for all n ∈ N_new do
12:        if n ∉ N_old then
13:            nodes+ ← nodes+ ∪ n
14:    for all e ∈ E_old do
15:        if e ∉ E_new then
16:            edges− ← edges− ∪ e
17:    for all e ∈ E_new do
18:        if e ∉ E_old then
19:            edges+ ← edges+ ∪ e
       return ⟨nodes+, nodes−, edges+, edges−⟩
```

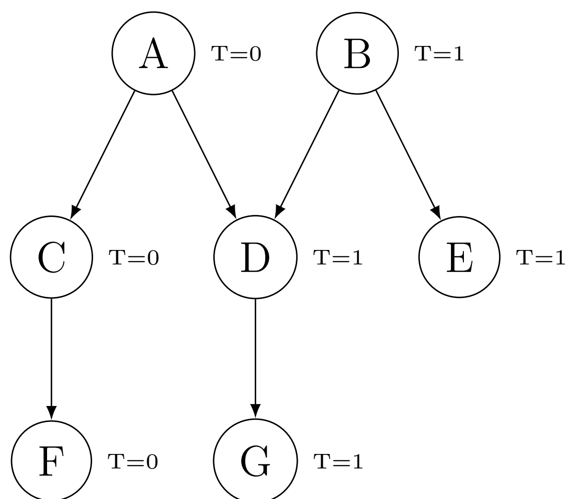

Figure 3.3 Example of a converged graph after an execution

the source and the node in question. If the removal is not in the transitive predecessors, it cannot impact the taint. Subsequently, if a part of the graph is removed, the impact on the taint will be limited to the successors of the removed part.

Another characteristic of a removal is that it can only reduce the value of the taint. Reducing is the action of changing the taint value from true to false. Considering the implication in Equation 3.9, an untainted node n implies that the node is a validator or that no predecessor p is tainted.

$$Taint(n) = False \implies n \in validators \lor \nexists p \in predecessors(n)(Taint(n) = True) \tag{3.9}$$

If the node is a validator, the taint will never be true regardless of how much the predecessors change. If no predecessors are tainted, the removal of one of them cannot taint their successor. Furthermore, the removal of any of these predecessors will always result in an untainted successor.

If the successor n of a removed edge is tainted, it is because at least one predecessor p was tainted or that n was a source (Equation 3.10).

$$Taint(n) = True \implies n \in sources \lor \exists p \in predecessors(n)(Taint(n) = True) \tag{3.10}$$

The removal of a predecessor will reduce the taint if it was the only tainted predecessor; otherwise, it will stay the same since a tainted predecessor still exists. Also if it was a source, the removal of a predecessor will not remove the taint, and it will remain. Hence, the removal of an edge can only retain or reduce the taint in a graph. If we remove all the edges and nodes that are to be removed because of an incremental update, the overall computation will be monotone.

The update of the taint caused by removals is implemented in Algorithm 5, which also uses this strategy. It is nearly identical to the regular algorithm. This algorithm is designed to operate on the new iteration of the program and the parameter $IDFG$ corresponds to this new iteration. It no longer has the removed nodes and the removed edges in the graph itself, but with the help of the stored results and Git, we can still retrieve them to operate our algorithms. This is important for the initialization of the work list. We only push the nodes when their taint could possibly be reduced (i.e., the successors of removed edges that still exist in the new version).

The line 3 iterates the removed edges and 4 ensures we do not push a node that no longer exist in the current version. As previously mentioned, we want to only handle the removal of parts of the graph; it is important to momentarily ignore the new edges in the graph. If we did not ignore the new edges, we could no longer guarantee that the taint can only be unchanged or reduced. If the algorithm could also increase the value of the taint, it would no longer ensure the monotonicity of our algorithm, and it may never reach a fixed point. The conditions at line 15 and 21 ensure that we ignore them. Finally, we do not have to report newly tainted sinks since it is impossible when only reducing taint.

Algorithm 5 Incremental reduce fixed-point taint analysis

1: **procedure** ReduceAnalysis($idfg, nodes^-, nodes^+, edges^-, edges^+$)
2: $\quad worklist \leftarrow List()$
3: $\quad$ **for all** $\langle p, s\rangle \in edges^-$ **do**
4: $\quad\quad$ **if** $s \in idfg$ **then**
5: $\quad\quad\quad worklist.insert(s)$
6: $\quad$ **while** $\neg worklist.empty()$ **do**
7: $\quad\quad c \leftarrow worklist.pop()$
8: $\quad\quad$ **if** $c \in sources$ **then**
9: $\quad\quad\quad inTaint \leftarrow True$
10: $\quad\quad$ **else if** $c \in validators$ **then**
11: $\quad\quad\quad inTaint \leftarrow False$
12: $\quad\quad$ **else**
13: $\quad\quad\quad inTaint \leftarrow False$
14: $\quad\quad\quad$ **for all** $p \in predecessors(c)$ **do**
15: $\quad\quad\quad\quad$ **if** $\langle p, c\rangle \notin edges^+$ **then**
16: $\quad\quad\quad\quad\quad inTaint \leftarrow inTaint \lor taint[p]$
17: $\quad\quad$ **if** $inTaint \land \neg taint[c]$ **then**
18: $\quad\quad\quad taint[c] \leftarrow inTaint$
19: $\quad\quad\quad$ **if** $c \in sinks$ **then**
20: $\quad\quad\quad\quad$ **Report tainted sink**
21: $\quad\quad\quad$ **if** $\langle c, s\rangle \notin edges^+$ **then**
22: $\quad\quad\quad\quad worklist.insert(s)$

If we apply this algorithm to Figure 3.3 with the removal of nodes A and B and their respective edges, we will obtain a graph like the one in Figure 3.4. This will simply result in a graph where all the taint values have been reduced to untainted since B was the only source of taint. The only nodes unchanged would be C and F because they are not reachable successors of B. C will be processed because of the removal of A but not its successor F since C did not update its taint. In this scenario, F would be the only node excluded by our algorithm.

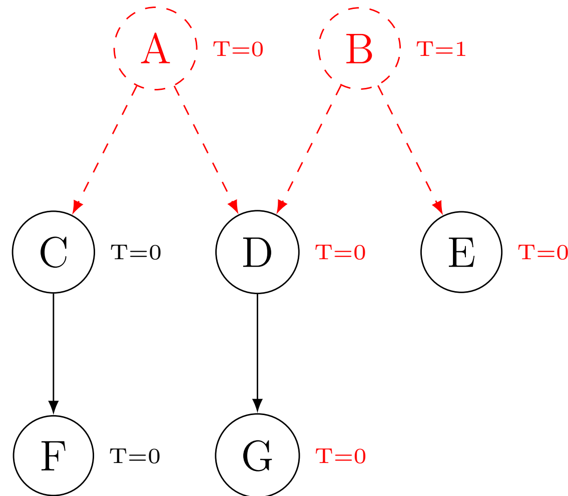

Figure 3.4 Example of a converged graph after the removal of nodes and edges

Like the removal of an edge, the addition of a new edge or a new node may only impact the nodes that are reachable by using the successor's edges transitively. Again, considering the equation of taint (3.1) on any node, the addition may only have an impact if it creates an unprotected path.

Similar to the removal of an edge or a node, the addition of a new element to the graph can only augment the taint values in the graph. It can only augment the taint from false to true. As described previously in Equation 3.10, a node is tainted only if it is a source or it has a tainted predecessor. The addition of a new predecessor cannot change the fact that a tainted predecessor exists.

Finally, if a node is not tainted, the addition of a node or an edge cannot further reduce the taint value. As said in Equation 3.9, an untainted node is either a validator or did not have any tainted predecessor. If the node is a validator, the addition of a predecessor changes nothing, regardless of whether it is tainted. If there was no unprotected path previously, the taint will be equal to the taint of the new predecessor. If the new predecessor is untainted, it means there is still no unprotected path that can reach this node. If the predecessor is tainted, it means that an unprotected path now exists.

By using these together, if we only compute the new edges and new nodes that will be created because of an incremental update, the overall computation will be monotonous and only increase the taint in the graph. The update of the taint for the addition of new edges and new nodes is implemented in Algorithm 6. Compared to the baseline algorithm, we only have to initialize the taint value of the new nodes and push the nodes possibly impacted by the addition of edges. These nodes are the successors of new edges and the new nodes

themselves. This ensures that a new source without predecessors will be computed and correctly propagated. The default taint of the new nodes is set to untainted since the other choice would break the monotonicity.

Algorithm 6 Incremental augment fixed-point taint analysis

1: **procedure** AUGMENTANALYSIS($idfg, nodes^-, nodes^+, edges^-, edges^+$)
2:     **for all** $n \in nodes^+$ **do**
3:         $taint[c] \leftarrow False$
4:     $worklist \leftarrow List()$
5:     **for all** $n \in nodes^+$ **do**
6:         $worklist.insert(n)$
7:     **for all** $\langle p, s\rangle \in edges^+$ **do**
8:         $worklist.insert(s)$
9:     **while** $\neg worklist.empty()$ **do**
10:         $c \leftarrow worklist.pop()$
11:         **if** $c \in sources$ **then**
12:             $inTaint \leftarrow True$
13:         **else if** $c \in validators$ **then**
14:             $inTaint \leftarrow False$
15:         **else**
16:             $inTaint \leftarrow False$
17:             **for all** $p \in predecessors(c)$ **do**
18:                 $inTaint \leftarrow inTaint \vee taint[p]$
19:         **if** $inTaint \wedge \neg taint[c]$ **then**
20:             $taint[c] \leftarrow inTaint$
21:             **if** $c \in sinks$ **then**
22:                 **Report tainted sink**
23:             **for all** $s \in successors(c)$ **do**
24:                 $worklist.push(s)$

Starting from the results of Figure 3.4, we add a new tainted node H and link it to the existing node D. We obtain the final computation as given in Figure 3.5. In this example, only the nodes H, D, and G will be processed and updated. The other nodes are not reachable from H and do not require validation. Using this brief example, a complete full analysis would have given the same results. C, E, and F would have been set to untainted, while D, G, and H would have been set to tainted.

By successively using Algorithm 5 and Algorithm 6, we iteratively compute the taint of every change until we reach a fixed point to obtain the updated taint of our analyzed application.

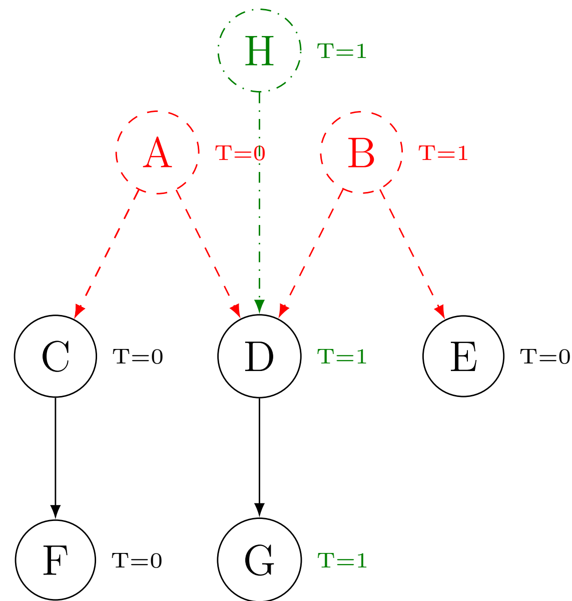

Figure 3.5 Example of a converged graph after the addition of a node and an edge

3.2.6 Research Questions:

RQ1: Given a set of version pairs found in existing projects, what is the ratio of nodes processed between an incremental analysis and a baseline analysis to compute the newest version? Additionally, with respect to the amount of source code changes and the taint impact of changes.

We assume a commit is generally a small increment from the last version. In most cases, modifications made by a developer should be small compared to the total size of the system. Our strategy would become advantageous when we only process nodes impacted by the changes in the flow graph, but how does this impact compare to our baseline analysis?

RQ2: How is incremental analysis faster in practice than a global analysis?

Even if the impact in terms of the number of nodes processed is low, the speed of our incremental analysis may be slower than a global analysis. The algorithms have additional instructions that may negatively impact the speed of the analysis. In practice, how does the computation time compare for the two approaches?

3.3 Differential Trace Generation

3.3.1 Trace Generation

With our taint analysis approach, developers will know which functions are currently vulnerable, but they may have difficulty finding the root cause of why a function in the code is vulnerable. A root cause is flawed code that must be fixed to protect a vulnerable sink. We generate examples of unprotected paths to help developers identify this type of code. Traces are examples generated by our algorithms. In this section, we discuss trace generation and how change-based techniques can be used to our advantage to improve the process.

For a given sink, we could try to generate an example for all the unprotected paths. However, it would be impossible to compute without approximations due to the presence of cycles in the DFG. Even with approximations, developers will probably be overwhelmed by all the information [21]. If only one path is displayed, other unprotected paths may not be fixed immediately, and the developer will only find out in the next analysis, which might be annoying if the amount of time required to complete the analysis is lengthy. Another parameter to consider is the amount of time required to generate individualized traces based on the chosen trace generation strategy.

When displaying traces to the user, we must choose the most relevant ones to help the developer identify the root cause. Researchers have identified some practices that can help make a trace more meaningful for the user: (1) Shorter traces are more efficient than longer ones [17]; (2) The traversal of a call site for a polymorphic function should be avoided since it has a higher chance of creating false positives [20]; and (3) The overall number of functions traversed should be minimized to increase the locality of the trace [16].

All paths

To pseudo-generate all the unprotected paths in Equation 3.11, we must search for all the paths p from any source n_o to the sink n_i that does not contain a validator. This approach can generate an enormous number of paths and can be infeasible if we do not ignore some traces. In the generation process, we need to ensure that a node can only be present once in the path. Otherwise, the number of paths for any looping graph will be infinite. If the path A-B-C-B-D exists, and non-unique nodes are not imposed, the path A-B-C-B-C-B-D should also exist. In consequence, we could find an infinite number of paths for this brief example by adding the edges B-C-B repeatedly. To avoid this problem, we use the algorithm designed by Princeton for a simple generation of all the unprotected paths without cycles [8].

$$\forall p_{(n_o,\ldots,n_i)} \in IDFG\{n_o \in sources \wedge n_i \in sinks \wedge \nexists n_x \in p(n_x \in validators) \implies p \in AllTraces(n_i)\} \tag{3.11}$$

Shortest path

An easier approach than searching all paths would be to find the shortest unprotected path for each sink. The shortest path fits the predicate in the Equation 3.12. It is implemented as a form of breadth-first traversal but could be done differently [29]. The algorithm will start from the tainted sink and iterate over the predecessors to find the source. It will also remember the nodes traversed to reduce the number of nodes traversed. It will permit backtracking to the sink at the end and generate the shortest path. The algorithm 7 is used over the IDFG.

$$\forall p \in AllTraces(n_i)\{\nexists p_x \in AllTraces(n_i)(|p_x| < |p| \wedge p_x \neq p) \implies p = ShortestTrace(n_i)\} \tag{3.12}$$

Spanning tree

An alternative to the generation of unprotected paths is spanning trees. This approach can find multiple paths rather than just the shortest path and does not take as much time as an excessive all-paths generation. We define our spanning tree as a tree covering all the tainted nodes in the IDFG reaching a given sink. Additionally, there is no notion of weight in our graph. The spanning trees can be computed with Algorithm 8.

3.3.2 Differential Strategy

Like taint analysis, the exhaustive generation of traces can be improved with change-based approaches.

For the traces, we will use a differential analysis to ignore previously tainted sinks and only generate traces for newly tainted sinks. We give the developer only the new traces that are created by his recent changes. Although the details are not included here, it is possible to report the sinks that have been protected.

Due to time limitations, only the approach for shortest path generation has been tested, and future research is needed to investigate the other strategies.

Algorithm 7 Shortest Path Trace Generation

```
1:  function ShortestPath(sink)
2:      visited ← ∅
3:      from ← ∅
4:      q ← Queue()
5:      q.enqueue(sink)
6:      while q ≠ ∅ do
7:          c ← q.dequeue()
8:          if c ∈ sources then
9:              path ← {c}                                ▷ Ordered set
10:             while c ≠ sink do
11:                 c ← from[c]
12:                 path ← path ∪ c
                return path
13:         for all p ∈ predecessors(c) do
14:             if p ∈ validators(sink) then
15:                 continue
16:             if p ∉ visited then
17:                 visited ← visited ∪ p
18:                 from[p] ← c
19:                 q.enqueue(p)
```

Algorithm 8 Spanning Tree Trace Generation

```
1:  function SpanningTree(sink)
2:      visited ← ∅
3:      tree ← ∅
4:      s ← Stack()
5:      s.push(sink)
6:      while s ≠ ∅ do
7:          c ← s.pop()
8:          for all p ∈ predecessors(c) do
9:              if p ∉ visited ∧ taint(c, sink) = True then
10:                 visited ← visited ∪ p
11:                 tree ← tree ∪ ⟨p, c⟩
12:                 s.enqueue(p)
        return tree
```

To find new traces, we will use information from the incremental taint analysis. When a new vulnerable API is detected as insecure in the taint analysis, we compute the trace of this new tainted sink.

However, it can be done on a generic taint analysis with one drawback: the detection of newly-tainted sinks. In this case, we need to use the stored taint values and ensure that a sink is newly-tainted compared to the previous version of the analyzed application.

Differential trace generation is much easier to implement than incremental taint analysis. The generation of the traces themselves is not different from normal generation; the only difference is how it is triggered. In the normal strategy, we forward all the tainted sinks to the generation, while, in the differential version, we only send a selection of sinks, which are the new ones.

3.3.3 Research Questions

RQ3: Given a set of version pairs found on existing projects, how does the size of traces generated for a differential analysis compare to a global analysis when computing the newest version? Additionally, how does it relate to the amount of source code changes and the taint impact of changes.

Given the pair of versions found on existing projects, we want to investigate how much lower the number of differentially generated traces is when compared to the total number of actual traces found in an application with a global analysis.

RQ4: How is differential trace generation faster in practice than global generation?

Considering the number of traces saved, how time efficient is our differential strategy compared to a full trace generation?

3.4 Experimental Design

Our experimental design needs to execute incremental and differential analyses of multiple versions to address our research questions. Therefore, we selected two web applications, WebGoat and Roller, and analyzed 60-100 commits for each of them. For each commit, we executed both a regular full analysis and the change-based analyses. We measured both the time required and the number of nodes processed for the various steps of our analyses and compared them to the regular strategy to assess performance.

We created some python scripts for this experiment. With a repository Uniform Resource

Locator (URL) and two specific commit versions, a script will fetch the commits, forming a path from one version to another and ignore commits outside of the path. Such behavior can be achieved by using "*git log [...] –first-parent [...]*". This choice was made to facilitate our experiment; however, it is possible to analyze commits that are not subsequent.

The complete experimental design can be seen in Process 9.

Process 9 Generic experimentation process

1: Generate and store the data-flow graph of both versions.
2: Execute and store the full fixedpoint for the previous version.
3: Execute the git diff tool for the pair.
4: Using the diff and the graphs for both versions, map bi-directionally the data-flow nodes and edges between versions and find the removed and added nodes and edges.
5: Execute and store the incremental fixedpoint for the current version.
6: Execute the full fixedpoint for the current version and ensure integrity with the incremental results.
7: Using the fixedpoint, find the new tainted sinks in the new version and generate new traces for them.

The experiment was conducted on a Virtual Machine in the IBM environment. The specifications of the virtual machine are 16GB of Random-Access Memory (RAM) with two processors on an Intel Xeon E3-12xx v2 at 2.49 GHz.

For the purposes of our project, we limited the analyzable languages to Java. Although they would have been interesting to analyze, Android applications and applications using Java Development Kit (JDK) 1.8 or higher are not within the scope of our current project. Of the available open source projects, we limited our research to web applications that are more interesting for taint analyses.

For these reasons, we analyzed older versions of WebGoat, an insecure Java application made by OWASP and previously inspected within other research projects related to taint analysis [7, 27, 28]. We analyzed 100 commits between releases 5.4 and 6.0. Of these commits, 38 had changes that impacted the data-flow graph. In the 5.4 release, there were 24,608 lines of Java code according to Cloc software [1].

The other web application we analyzed is Roller. It is a blog application that supports a large number of users, groups, and blogs [3]. This project was chosen because it is a web application with security elements and is a full application rather than many small examples,

like WebGoat. We analyzed 70 commits between versions 5.1.1 and 5.12. Of these commits, 27 had changes that impacted the data-flow graph. The 5.12 release had 53,465 lines of Java code.

3.5 Results

3.5.1 Taint analysis

RQ1: Given a set of version pairs found on existing projects, what is the ratio of nodes processed in an incremental analysis and a baseline analysis when computing the newest version? Additionally, how does the amount of source code changes compare to the taint impact of changes?

For WebGoat, we measured the number of nodes processed by the two phases of the incremental algorithm, the reduction and the augmentation, and compared it to the number of edges changed in the IDFG for the corresponding version analyzed. These figures show how many nodes were de-queued in relation to the number of the edges that have been removed or added in the IDFG. They are the edges that are different in the IDFG compared to the IDFG from the previous analysis. These results have been compiled in Figure 3.6 for WebGoat and in Figure 3.7 for Roller.

These results reflect the impact of using incremental analysis compared to the baseline in terms of nodes processed. This impact is mainly caused by the algorithm, which handles the additions in the graph. By summing the number of nodes for the two phases together, we can determine the ratio for the incremental algorithm and a global analysis. The ratios for all the WebGoat and Roller commits analyzed are in Figure 3.8.

In this figure, each dot represent a commit. They have been ordered by the percentage of new edges over all the edges in the inter-procedural data-flow graph, or what we consider a percentage of change, as the name of the axis. This approximation corresponds to the number of new edges over the total number of edges in the new versions. In this section, when percentages of change is used in an axis for a diagram, it corresponds to this estimate.

The ratio of nodes processed heavily advantages the incremental analysis but grows with the percentage of changes. Considering the nature of our algorithms and the results, it is highly probable that a correlation exists between the ratio of nodes and the percentage of changes. A measurement of the Pearson correlation coefficient indicates a coefficient of 0.955 for the commits on WebGoat and 0.815 for Roller.

RQ2: How is an incremental analysis faster in practice than a global analysis?

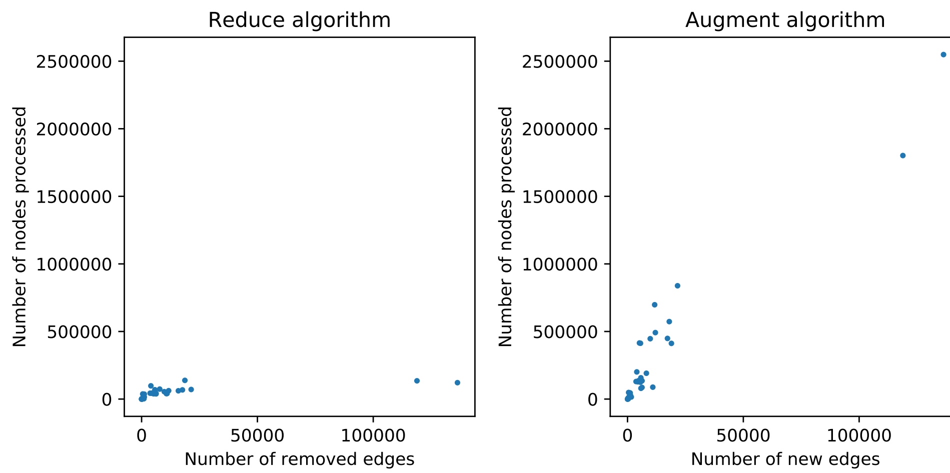

Figure 3.6 Impact on incremental analyses in relation to flow changes in WebGoat

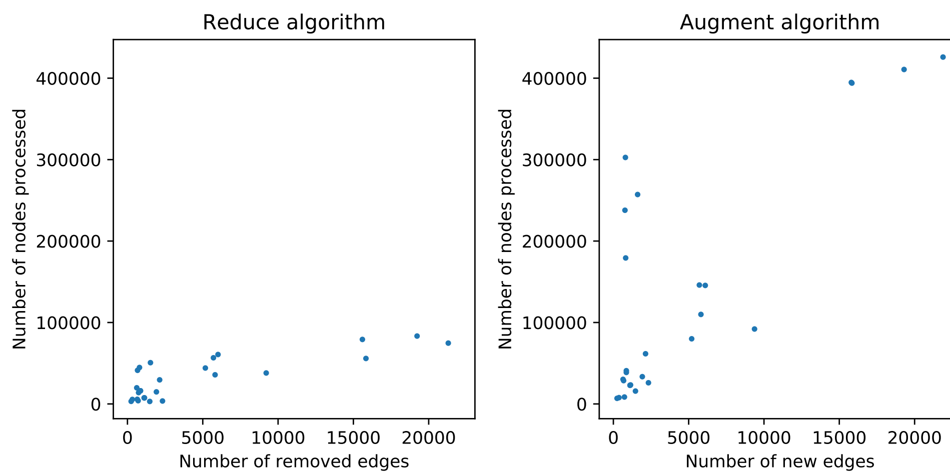

Figure 3.7 Impact on incremental analyses in relation to flow changes in Roller

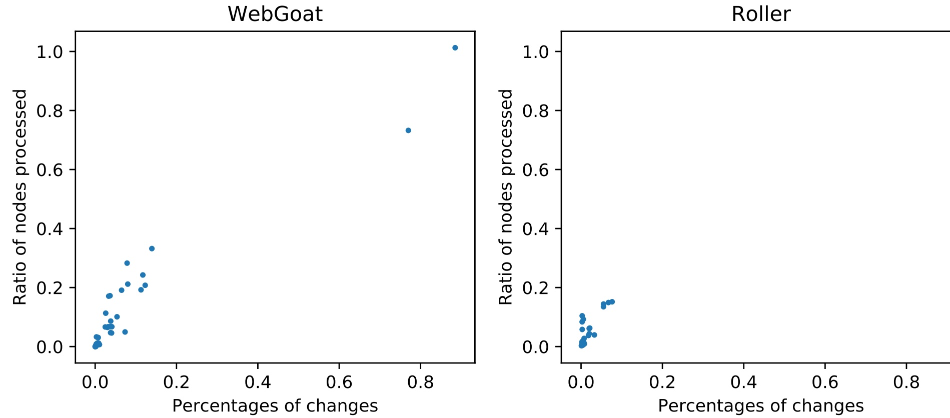

Figure 3.8 The ratio of nodes processed in the incremental taint analysis and a global taint analysis

In Figure 3.9, we report the timing results of taint analysis and global analysis for every commit analyzed on both projects.

Incremental analysis seems very effective for small commits. For the smallest ones with only a few edges changed, it take less than a second to update the taint values compared to nearly ten minutes without our approach. On both WebGoat and Roller, all the commits are below 20% of the data-flow edges changed according to our estimate, with only two exceptions. For all the commits, the time needed for an incremental execution is drastically lower than the regular execution, which is always in the range of 500-700 seconds for WebGoat and 600-800 seconds for Roller. A variation greater than three minutes is large. We found two possible factors that may explain this variation. First, between commits, the code changes, and the size of the graphs could possibly change enough to have an impact. Secondly, we are running on a cloud environment with a certain level of fluctuation. We have noted that depending on the day or time the analysis ran, there was a variation in the speed of our algorithms. This may explain why some outlier results on Roller are close to 100 seconds. A counter-measure would have been to run the analysis multiple times on every commit to statistically mitigate this issue. However, we were unable to do so due to time constraints.

For the two commits with a high ratio of changes on WebGoat, the time needed is close to the full taint analysis. It could indicate that the overhead for an incremental analysis is not significantly large, and the overall speeds of both algorithms are close. However, more investigation on the commits with a high ratio of changes would be needed to make this conclusion.

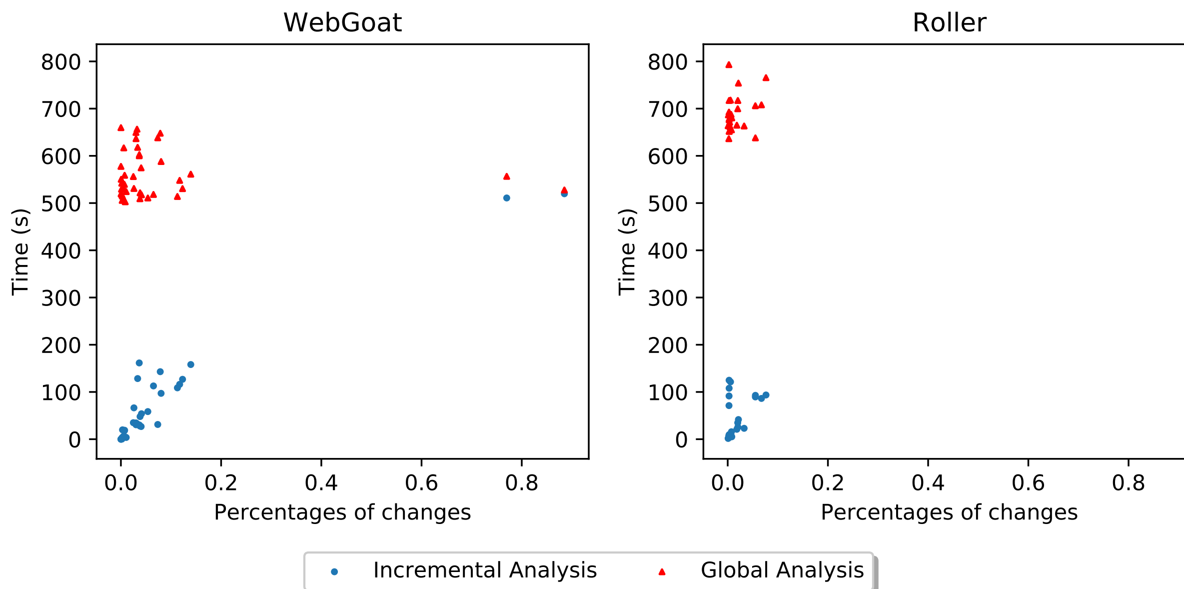

Figure 3.9 The performance of incremental taint analysis and global analysis as it relates to data-flow changes

When most of the commits are relatively small, the time reduction is quite evident for the incremental taint analysis. The reduced time represents, on average, only 12.87% of the full-taint analysis for WebGoat and 5.99% for Roller.

3.5.2 Trace generation

RQ3: Given a set of version pairs found on existing projects, how does the size of traces generated for a differential analysis compares to a global analysis when computing the newest version? Additionally, how does the amount of source code changes compare to the taint impact of changes.

The number of traces generated is presented in Figure 3.10 for our analyzed projects.

Since multiple commits do not generate any traces, we present the proportions of commits generating differential traces against the ones that do not generate new traces. The pie charts in Figure 3.11 present these proportions.

The proportions of commits without new traces are around 42% on WebGoat, while on Roller, it is closer to 60%. We suspect it is normal that a project focused on presenting vulnerabilities like WebGoat have a higher percentage of commits with new traces compared to Roller. Refactoring and adding vulnerabilities should be normal for this project. However,

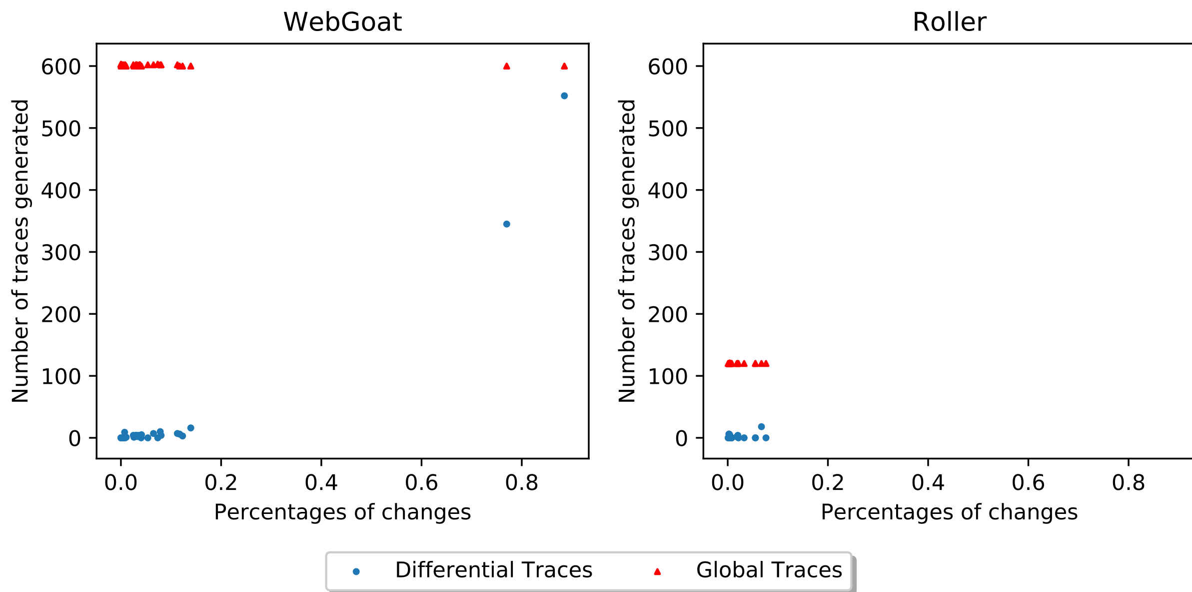

Figure 3.10 Distribution of traces generated by commit

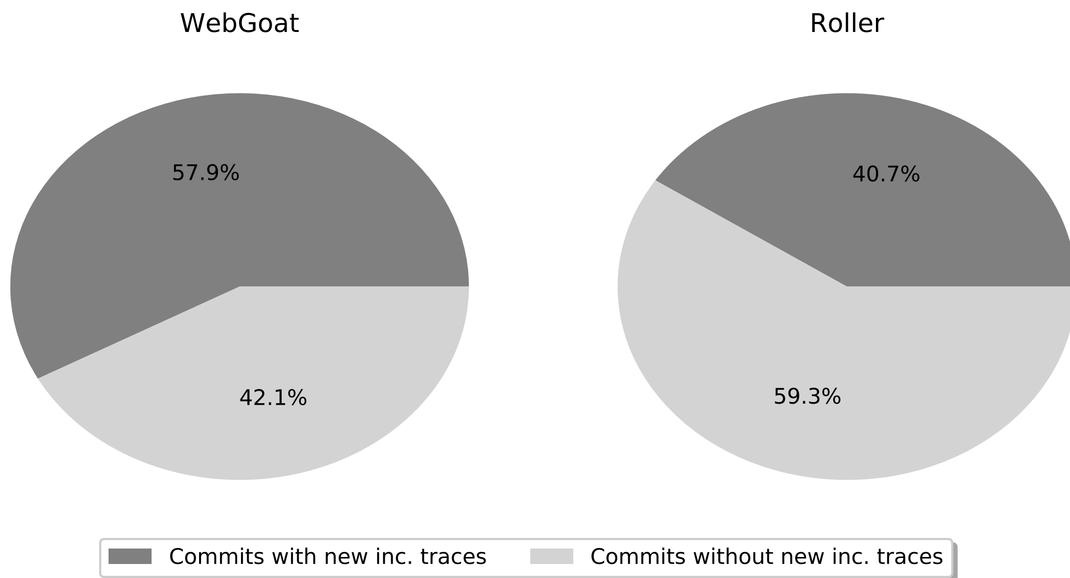

Figure 3.11 Proportion of commits with and without new differential traces

40% of commits with new vulnerabilities is a little high for Roller. There is a possibility that some of them are caused by files being renamed or moved, which triggers a re-generation of the same traces. When a file is renamed in our solution, all the nodes will be deleted and re-added. To address this issue, we should improve our differential tool to handle these situations. Additional evidence for this theory is the scattered points for the global analysis in Figure 3.10. The total number of traces does not change much between versions. For the versions that have changed or new traces, the overall number of traces differentially generated is fairly low compared to a global analysis. Only the two commits with a high percentage of changes have a significant number of traces generated.

RQ4: How is differential trace generation faster in practice than global generation?

The timing results for trace generation are presented in Figure 3.12.

It seems that most of the commits require an insignificant amount of time to generate differential traces. Only two of them take more than a minute, and they are the commits with a lot of changes in the flow graph. They do not seem to be the norm. However, it is surprising to see the points scattered so much for the global analysis of WebGoat. As seen in Figure 3.10, the number of traces generated for them is nearly the same, and we suspect the variation should not be that large.

3.5.3 Generic

Our inter-procedural strategy certainly had an impact on both performance and precision, but measuring them is difficult. In the current state, we cannot measure precision. However, it is still possible to evaluate the impact on the volume by measuring the number of contexts generated by our inter-procedural strategy.

Table 3.1 Metrics on the inter-procedural context strategy

Metric	Nb of units
Defined functions	1 679
Significant call sites	4 095
Contexts	8 425

We analyzed the results from WebGoat 5.4 in detail to extract the following metrics about the inter-procedural contexts in Table 3.1. We obtained the number of distinct defined functions, which is simply a function with a non-empty body in the scope of the project analyzed. If it is a function from an external library, and the implementation is unavailable, it does not count

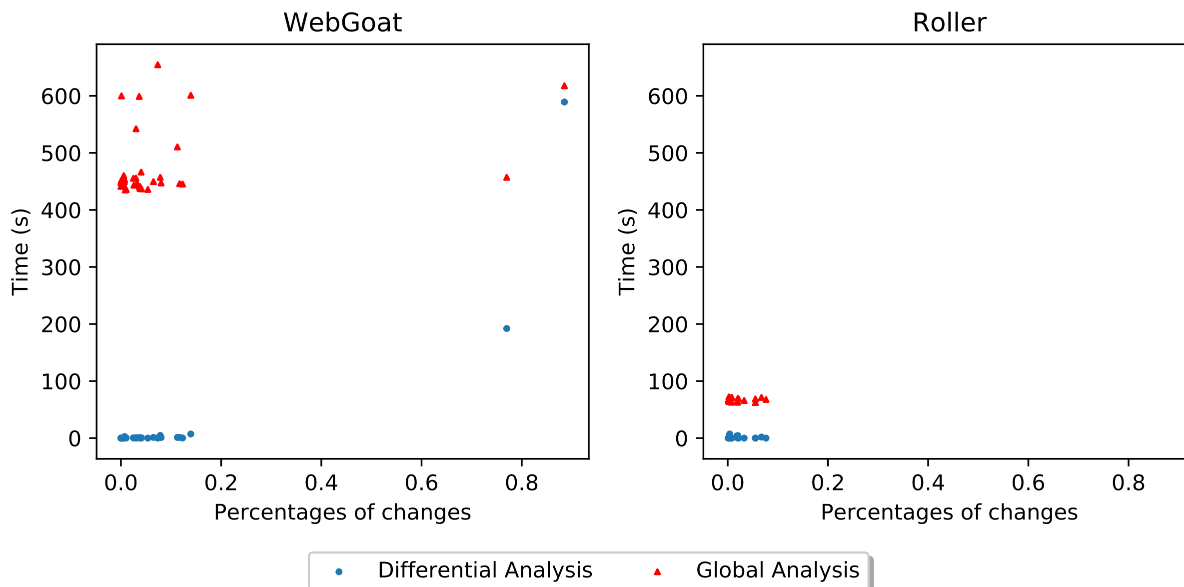

Figure 3.12 Performance of differential trace generation as it relates to data-flow changes

as a defined function. Although it is typically unimportant at the inter-procedural level, it is important at the intra-procedural level for our analysis. We have access to a database provided by IBM that contains details about the taint information of these functions. It can inform us regarding whether it is a source, a sink, or a validator, as well as whether it propagates taint from one parameter to another or to the return value.

The second metric is the number of significant call sites. A significant call site is an SSA operation that calls a function when at least one of the called functions is a defined function. They are the only call sites that can create new contexts, considering the defined functions mentioned previously. Thus, call sites that call at least one defined function are the only ones to be considered.

Finally, the third metric is the total number of distinct contexts generated by our strategy. We have roughly five times more contexts than functions, which means that a function is called from five different call sites on average and that our expanded graph is approximatively five time bigger than a full context insensitive strategy.

As already stated in Section 3.2.2, the use of equivalent classes of sinks and validators will drastically reduce the number of times our analysis needs to be run for a version. For example, in WebGoat 5.4, repeating the analysis 42 times would have required approximately 90 minutes, but an analysis on an equivalent sink takes approximately 130 seconds. In

comparison, the analysis performed with equivalent classes took 11 minutes by analyzing only 5 equivalent classes. It is a drastic time reduction, which is advantageous.

Our incremental and differential strategies reduce the overall computation effort. Like Figure 3.13 shows, the change-based approaches takes 90% less time for WebGoat compared to our regular analysis. This number is even better for Roller by taking 95% less time with respect to the commits we have analyzed. The change-based analysis timings does not include the time required to perform the analysis on the previous version. The goal is to compare how long a user has to wait before getting results if he already performed an analysis before making his changes.

3.6 Threats to validity

For this project, we only analyzed two open source applications. WebGoat is an artificial system designed for displaying security leaks. Thus, our results for this system may not be representative of the average project. However, since it contains a high number of vulnerabilities, we think it contains enough code and vulnerabilities to be interesting. Like we saw in Figure 3.10, many more traces are generated in WebGoat than in Roller, even if it is a smaller system in terms of the number of lines of code.

We have analyzed a total of more than 100 commits. On these commits, there is a range of changes, including no changes and tiny adjustments, as well as major renaming and refactoring. We think that a new system would not act in a completely different manner than what has been seen in this paper. The importance of the gains will vary but probably not enough to threaten the performance gains of our analyses. Due to the nature of our algorithms, the possible outcomes are quite limited, and other systems should only reinforce our results.

It would have also been quite interesting to analyze an application of a much larger scale than WebGoat or Roller. Since our analyses are considerably more bounded by the number of changes than by the size of the application, we think the performance of our algorithms would be even better on larger systems.

Our analysis is not CFG-centric like most research projects in the sense that our graph is not represented at the granularity of CFG statements; it uses data-flow elements. We are mainly focusing on a data-flow approach, which may give different results than on a regular system since the graph structure is different. The algorithms must be adapted to work on other projects. The economy reduction could differ in other frameworks. However, the algorithm itself is still very generic and could easily be adapted to other data-flow problems, primarily the ones based on the standard framework described in the book *Compilers: Principles,*

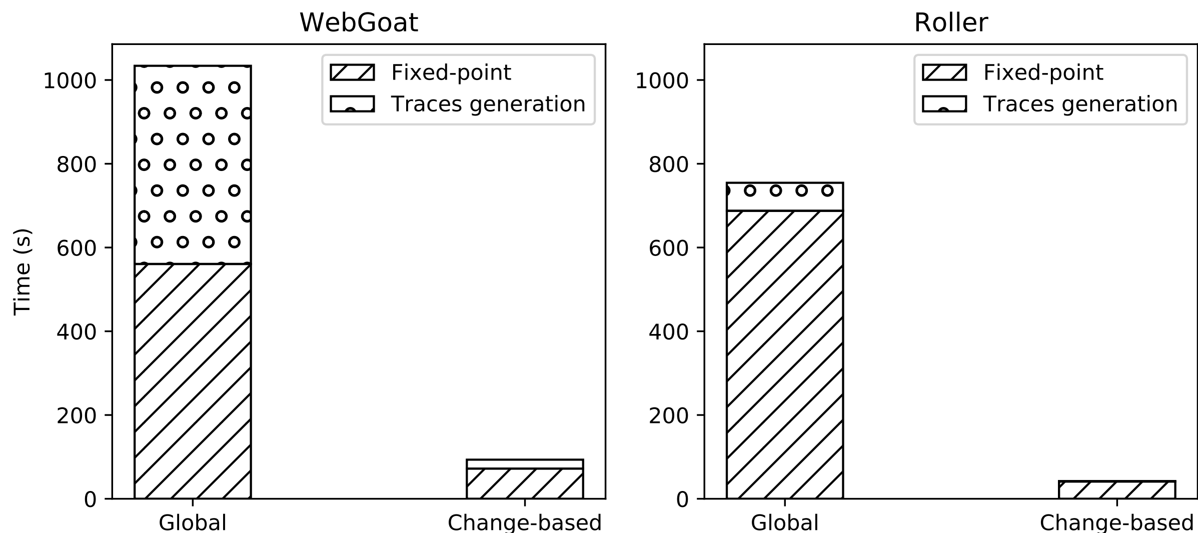

Figure 3.13 Comparison of mean time required for a regular analysis compared to our change-based approaches

Techniques, and Tools (2nd Edition) [10].

3.7 Related work

Many researchers tackle the speed issue by defining strategies to reduce computational effort. It is possible to do so but at the cost of using more memory or sacrificing precision. However, some projects are artificially reducing computation time by changing the order of computation [14]; the most relevant results display early while the remaining results are still computing.

Projects like Taint Analysis for Java (TAJ) and Andromeda work mainly on the principle of laziness or on-demand [27, 28]. They make taint analysis faster by only computing certain aspects, such as the taint or call-graph on-demand. This is no longer eagerly computed, and they can save time since some parts are unused.

While Andromeda has a section on incremental analysis, it is far from being the main point of interest: lazy-driven computation. This project works with an invalidation strategy for taint analysis. They delete the results of the previous taint analysis on the methods which are impacted by changes. Although we do not know the details of their exact strategy, we think we have a better solution by being more granular, which reduces the amount of re-computation.

An interesting project to compare with ours is Reviser by Steven Arzt and Eric Bodden [11]. Their goal is to make the static analysis engine faster by using incremental techniques. However, they are doing it for inter-procedural, finite, distributive subset (IFDS) and inter-procedural distributive environment(IDE) problems, which have specific ways of defining the terms problem and inter-procedural [23, 26]. The projects are similar, but we are not using the same framework, and the graphs we are operating with are different.

As their goal is fundamentally the same, we can expect similarities between our approach and theirs. We are both doing context-sensitive flow-sensitive inter-procedural analyses with transfer functions. Still, their approach has been generalized to multiple problems, for example, reaching definition and uninitialized variables, while we are specializing in taint analysis. However, we believe our strategy can be generalized to many more problems since the core of our algorithms are not dependent on the special characteristics of taint.

Excluding the frameworks, there are still important details separating us from the strategy used by Reviser.

To define the starting nodes for their algorithm, they need to find the safe predecessors of their changed nodes, which are nodes safe from being affected by a change. In comparison, we push the changed nodes or the direct successors of flow changes as starting nodes.

To ensure the monotonicity of their approach while maintaining precise results, they are resetting the old analysis values on the first visit of a changed node. If every analysis value restarts from the bottom of the semi-lattice and repeats the partial order, they will surely reach a fixed point. Our approach is different since we update the values without doing any sweep of the old values. The trade-off is that we may have to make multiple passes on a node, while their approach does not if the number of predecessors is less than two.

3.8 Conclusion

We have presented novel change-based approaches for taint analyses. We incrementally update taint to correspond to the changes in the source code. While updating the taint, we detect new vulnerabilities and generate traces for them. We think it could enable developers to better understand effects caused by changes in applications.

By grouping sinks, we reduced the amount of time required to find vulnerabilities by a factor of 8. Furthermore, by using changed-based approaches, we greatly reduced the computation time by an average of 90 to 95% on our analyzed projects.

One of the worst cases encountered, when nearly all the code changed for WebGoat, we found that it takes close to the same amount of time as a regular approach. These cases could be

corrected by a more effective differential tool to examine pairs of versions.

These results possibly open the door to a better way to integrate static analysis techniques into a developer workflow due to the reduced time required for analyses with small code changes. It would be possible to include our improvements in taint analyses performed on integration servers to efficiently analyze each commit submitted. Additionally, with our incremental taint analysis, it would be possible to integrate the strategy in an IDE and expect an interactive experience with the developer. With granularities close to an instructional level, incremental changes should have an impact low enough to make this possible.

This approach is easy to implement, and we hope to see it implemented and tested on problems other than taint analysis.

3.9 Acknowledgements

This project has been made possible due to a collaboration with and sponsorship by the IBM Centre for Advances Studies Canada, who also gave us the opportunity to implement and test our algorithms using the software IBM Security AppScan Source. We are especially grateful to Vio Onut from IBM for his support in this project.

We would also like to thank both the Natural Sciences and Engineering Research Council (NSERC) and Fonds de recherche du Québec – Nature et technologies (FRQNT) for sponsoring this project.

CHAPTER 4 TAINT REACHABILITY OPTIMIZATION

4.1 Introduction

In this chapter, we present one optimization implemented for trace generation. By making this analysis faster, we take one step closer to creating an interactive experience for developers. The previous chapter discussed how to generate traces and proposed algorithms to do so. However, these algorithms visit more nodes than necessary to reach their results. It is possible to cut parts of the graph while searching for the shortest path, but consequently, the additional logic required to do so could reduce performance. This chapter presents the tactics used and discusses the effects, in practice, on the efficiency and performance of this modification of the shortest path algorithm.

4.2 Approach

The shortest path and all-paths algorithms are search strategies that find paths in the graph. As detailed in Section 3.3, we start a search from a tainted sink and transitively search for a reachable source. The spanning tree is a third option, but it will not be discussed in the current chapter. We will focus only on the shortest path.

Per the definitions of taint and of traces, every node of a trace must be tainted, as mentioned in the equations in Section 3.3. Every node of a trace is reachable from the source, and no validator exists in the path to block the taint. Yet, our algorithms do not consider this property in their search and explore every predecessor independently.

For the following reasons, the choice to explore only the tainted nodes will not perturb the resulting paths. An untainted node has two possible implications, either that it is not reachable from a source or that it is validated. If the shortest-path algorithm performs as we defined and explores a part of the graph that has not reached a source, it will simply waste its time and never find a trace. In the case of a validator, if the algorithm reaches the validator, it will block the search, and it will never trigger the generation of a trace.

Consequently, if we follow only the tainted predecessors and not all, like we did in the previous chapter, we are certain to still reach a source and find a path. This will simply lead to less nodes being explored.

To enforce that we only visit tainted nodes, we must perform a check on every node traversed by the algorithms. Since this is an additional instruction, it may slow down the generation.

In the worst case, where every single node is tainted, the number of nodes visited will be the same, but we will have performed many more instructions. In the context of taint, we will evaluate if the size of the graph cut is large enough, in comparison to what is explored, to be worth the effort in terms of performance.

4.3 Research Questions

RQ: What is the ratio of nodes reduced by using the taint reachability optimization in comparison to not using it?

We would like to investigate how effective taint reachability optimization is on our trace generation. We hypothesize that it will reduce the volume of nodes visited, but we do not know in which proportions.

RQ: In practice, how much faster is the trace generation with the usage of taint reachability optimization?

We may reduce the number of nodes explored with optimization. However, this is at the cost of an additional conditional check on every node iterated during the trace generation. Considering this, we want to investigate if the time required will be higher or lower than without utilizing this approach.

4.4 Methods

We implemented the shortest path algorithm defined in Algorithm 7 for this experiment. Two versions were implemented inside the software AppScan, as described in the previous chapter. The baseline version acted exactly like the algorithm. However, the optimized version modified the predicate on each line 16 to ensure that the predecessor was both not visited and tainted for the corresponding sink. The modified algorithm can be seen in Algorithm 10.

We analyzed WebGoat 5.4 for our results. We directly measured the number of nodes processed by the shortest path, which corresponded to the number of items "dequeued" by the algorithm. Also, we measured the time taken for the function corresponding to the algorithm. The measurements were performed for the traces of every vulnerability found by the taint analysis.

The experiment was performed on the same system as the one used in the previous chapter. The system is a virtual machine hosted on a cloud system. The specifications of the virtual machine are 16GB of RAM with two processors on an Intel Xeon E3-12xx v2 at 2.49 GHz.

Algorithm 10 Shortest Path Trace Generation with reachability optimization

```
1:  function ShortestPath(sink)
2:      visited ← ∅
3:      from ← ∅
4:      q ← Queue()
5:      q.enqueue(sink)
6:      while q ≠ ∅ do
7:          c ← q.dequeue()
8:          if c ∈ sources then
9:              path ← {c}                                  ▷ Ordered set
10:             while c ≠ sink do
11:                 c ← from[c]
12:                 path ← path ∪ c
                return path
13:         for all p ∈ predecessors(c) do
14:             if p ∈ validators(sink) then
15:                 continue
16:             if p ∉ visited ∧ taint[p] then
17:                 visited ← visited ∪ p
18:                 from[p] ← c
19:                 q.enqueue(c)
```

4.5 Results and Discussion

Figure 4.1 reports the number of nodes traversed by the shortest path algorithm. On the X-axis, we show the traces for all the tainted sinks, which have been ordered by the number of nodes traversed without the use of the optimization. The two bars represent the optimized approach compared to a baseline, which explores everything independent from the taint values, as defined in the previous section.

For all the traces, the number of nodes traversed was lower. On average, we explored 84 percent of the nodes traversed by the baseline. Twenty-two of our traces did not have any improvement, and in the best case, the number of nodes visited was reduced to 32 percent.

Figure 4.2 reports the time it took to generate every trace of the shortest path. The traces were ordered by the time taken for the baseline strategy. Generation takes, on average, 6 percent more time with the use of the optimization.

Even if we visited less nodes than in the baseline solution, the overhead of checking the taint seems to be too big for what its worth. We expected the ratio of nodes cut by the strategy to be better than the existing 16 percent, which is not enough to give us the performance increase we wanted.

In the context of shortest path, we did not visit all the possible paths, only what was needed to reach a source. The algorithm stopped when the shortest trace was found. In other, more expensive approaches, the performances may be different. If we explored everything in reach of the sink, the impacts of cutting a part of the graph could be much greater.

We suppose that the all-paths algorithm, which could not be implemented for this experiment, would give the best improvements. It is the slowest algorithm, and it does not scale well with the size of the graph. By reducing this graph, we may get better performances, compared to the shortest path.

4.6 Threats to Validity

For this experiment, multiple factors affected the quality of our results. 600 traces were studied. However, they all come from the same version of WebGoat. To increase our confidence in our results, we need to drastically increase the number of versions and projects analyzed. Increasing the variety of benchmarks would make us more confident that the reachability optimization is worth or not. Also, by repeating the experiment we mitigate variability factors like the caching of the operating system, the caching in our framework and the workload of the virtual machine.

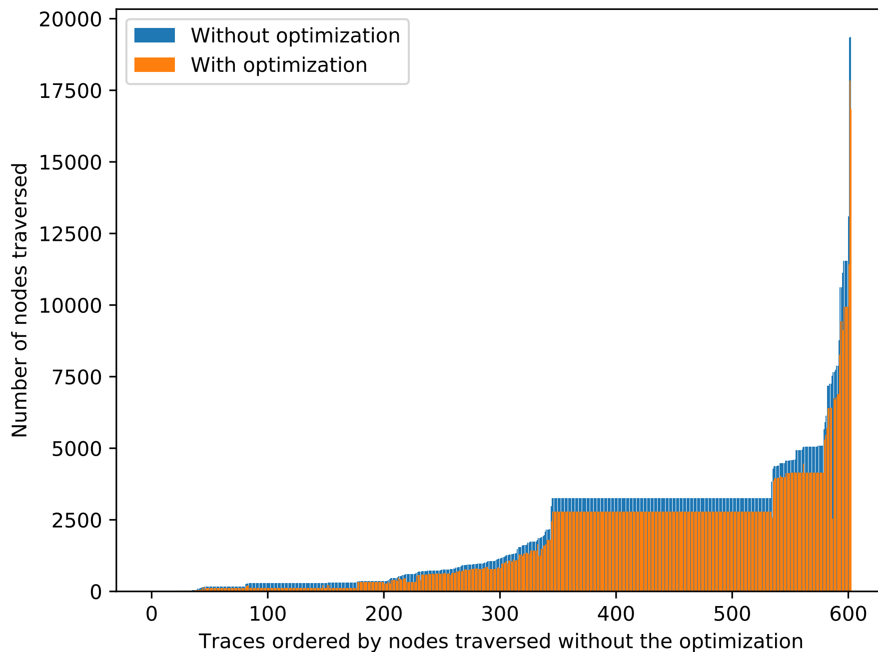

Figure 4.1 Histogram of nodes traversed for shortest path traces generated with and without the taint reachability optimization on WebGoat 5.4

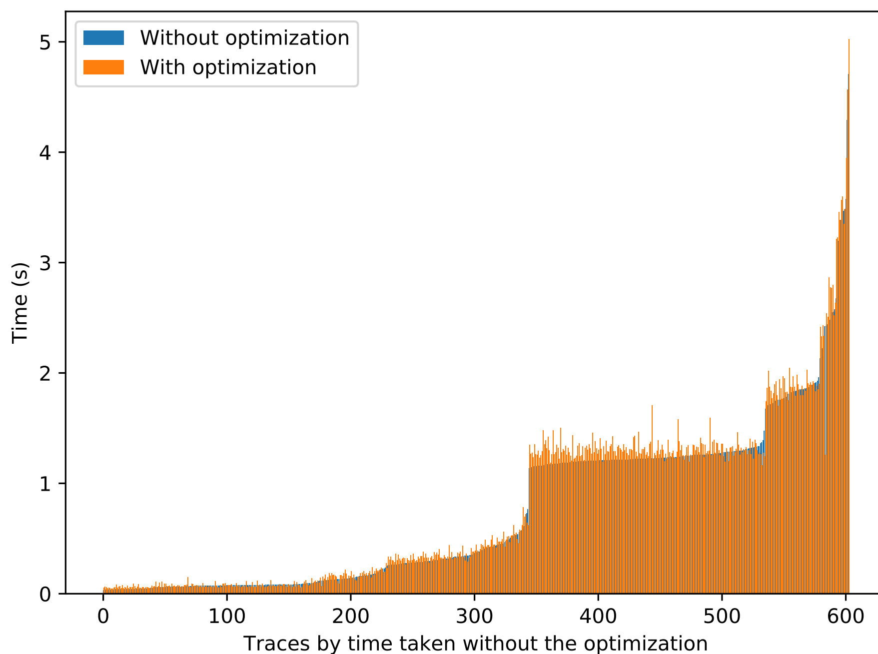

Figure 4.2 Histogram of time for shortest-path traces generated with and without the taint reachability optimization on WebGoat 5.4

4.7 Conclusion

In the situation of WebGoat 5.4, our optimization is not useful in the context of generating the shortest path for presenting a trace. It reduced the size of the graph explored by a margin of 16 percent, but it was slightly slower and took 6 percent longer. More research is needed to find out if it is unique to WebGoat or more generalized. Since WebGoat is an application made to be vulnerable, it can have more taint in the data-flow graph compared to an average application. Maybe other applications would perform better with our optimization. Also, we should investigate other implementations of the reachability check and see if it is possible to make it more effective.

CHAPTER 5 GENERAL DISCUSSION

In this chapter, we will present a general discussion of the overall results of this thesis. We will briefly explore our research objectives to complement the discussions in the main article. An examination of the possible impacts of our techniques on the developer workflow will follow.

5.1 Discussion of Research Objectives

Our main research objective was to reduce the computation time of taint analysis. We developed an incremental approach that updates the taint values based on the changes found in the source code. By doing so and analyzing two open-source projects, we reduced the time required by a significant margin. The time was reduced by an average of 90 percent on WebGoat and 95 percent on Roller.

By looking back at the results from our incremental taint analysis in Section 3.5, we observed some interesting results. Our analysis seems to perform well because the changes are relatively small in the majority of the commits. Most of the changes occur on under 20 percent of the source code. By looking at the two exceptions with a high percentage of changes in WebGoat, we can suppose that our analysis would perform worse if the changes were higher on every commit. Thus, it would be to our advantage to improve our algorithm mapping the data-flow graph between the two versions to correctly handle the changes in names and directories. This change should only make our analysis faster and not affect the results.

We would also like to discuss the variability seen in Figure 3.9 for the global analysis. On both Roller and WebGoat, we can see the global analysis varied within a range of roughly 200 seconds. However, we must be careful when looking at this graph since it does not show the number of lines of code. As a consequence, two nodes with a similar percentage of change can correspond to two other versions, with one being much bigger than the other in terms of the size of the codebase. Another factor to take into account for this variability is the usage of a virtual machine for the computation. The analysis was run inside a cloud system, which may cause a certain variation on in the performance of the working environment itself.

Our second research objective was to reduce the computation time of trace generation. Using the taint analysis results, we generated only the new traces and, thus, avoided much of the computation normally needed. In the results, we saw that, for the vast majority of the commits, the number of new traces was small compared to the total number of traces. This

worked well because we recomputed the traces only by looking at the changes on the sink. If the sink did not change, we did not try to regenerate the traces. An issue with our approach is that we will not update traces that could have a new path with the recent changes, but the sink did not change. For example, if the developer tried to fix the tainted sink but failed at doing his correction. It is unfortunate, but the cost in terms of performance would be higher if we tried to keep the old trace in memory and used it as a comparison to regenerate. One possible workaround would be to give the user the possibility of generating traces on demand. In such a scenario, he or she could generate the missing trace in a short amount of time.

5.2 Integration in the Developer Workflow

On the vast majority of the commits we have analyzed, the time needed for computing the taint has been reduced significantly. With such improvements, we will speculate what could be the impacts of integrating our improvements in the developer workflow. In the following sections, we will discuss why our strategy may be beneficial and what challenges are left to use it in certain situations.

5.2.1 Integration System

Some open-source projects already use static analysis on their code. For example, the integration tool Travis [1] already allow the usage of third-party static analysis tools on the analyzed projects. However the usage of these tools is usually limited to the users since the organisation behind the analysis cannot afford to run so many analyses. It may not be possible to run an analysis on every commit sent to the repository. As an example for Coverity Scan, frequent analysis is not recommended, and they have a hard daily limit [2]. On projects with more than a million lines of code, it is possible to run a scan only once per day. It is expensive to run analyses and it is certainly a challenge to offer it as a service to open-source projects. We do not know how they implement their static analyses. Assuming that they are not already using a certain form of incremental strategy, by implementing a change-based approach to their analysis it may be possible to reduce the time required for each scan.

For the developers of the analyzer, this means increasing the number of scans done in a day. They could reduce their cost of operating this service, or they could permit projects to increase the frequency of their scans. Ideally, developers could scan their project at every commit and obtain quicker results.

[1] `https://travis-ci.org/`

[2] `https://scan.coverity.com/faq`

The main trade-off would be a possible increase in required disk space. An average analyzer will not necessary store all the intermediate data computed during an analysis. Many of the data is temporary and can be recomputed from scratch. However for our incremental scanner to work in a integrated system, it must permanently store some data between analyses. At the very least, the system must keep old versions of the data-flow graph and all of the old taint results.

If we are developing a system optimized for only one project, we could try to avoid storing the analysis results on disk and keep it alive in the RAM. However, a full analysis would be needed every time a new instance of the analyzer starts, and the instance would need to stay alive as long as possible. When it comes to a system analyzing many projects, it would probably not have enough memory to keep all the instances alive with their respective intermediate results, and persistence would be required. This action would increase the time spent by the analyzer. Eventually, not only the taint analysis, but all of the other data structure computed like the CFG, DFG and the callgraph could be incrementally updated to maximize the performance of our integration system.

5.2.2 Interactive Developer Environment

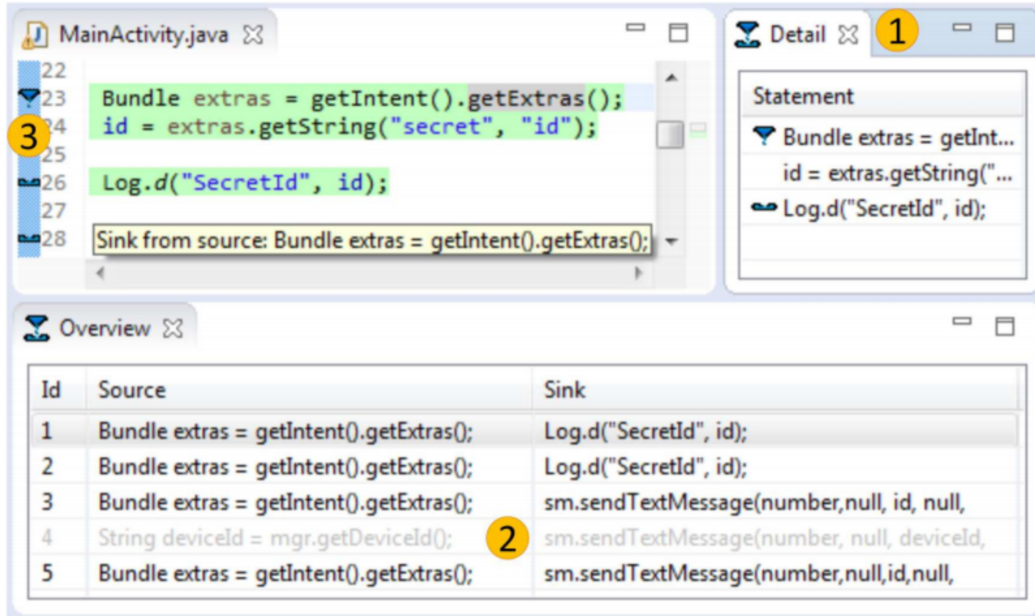

Figure 5.1 Example of the integration of the taint analysis tool Cheetah [14] in a IDE.

We imagine that such an integration in the developer environment could go even further. It may be possible to add our improvements to analyzers that are fully integrated in developer

environments. The project, Cheetah [14], already does that through an Eclipse plugin. As a visual example, we added a screenshot of Cheetah taken from their article in the Section 5.2.2. Interestingly, one of their future goals is to make their taint analyzer incremental. Such an analyzer could run every time a file is saved or a build has been completed. As long as the source code can be compiled, a quick scan may be possible.

For our taint analysis tool, it would be primordial to incrementally generate every data structure. If the generation of the AST, CFG, and DFG and the resolution of virtual calls is not incremental, the analysis would still take too much time. For our analysis itself, with an average speed of 4,000 to 5,000 nodes per second, it is not difficult to imagine that some lines changed could be analyzed in less than a second. This is already the case for the smallest commits in WebGoat, where some commits have only one or two lines of code changed, and our analyzer solved them in around 100 milliseconds. If we had incremental generation of other data structures, it would certainly increase the time it takes for a developer to get results. By using an approach similar to Cheetah [14] and making the scan on a worker thread, the taint can be computed without hindering the work of the developer and present the results as they come.

As in integrated systems, this improvement will increase the memory needed by the environment and may be problematic if the developer does not have enough RAM available. By adding more mechanisms, it could be, however, mitigated. We would need to add another level of complexity and incorporate a caching system to have only a subset of the graphs loaded at the same time.

CHAPTER 6 CONCLUSION AND RECOMMENDATIONS

Software projects usually evolve in an incremental fashion, gradually iterating and changing to become the software they are now. It is at the advantage of static analyzers to utilize this development property, which we did in the context of taint analysis.

As a reminder, our motivations for this dissertation are: (1) to help developers find the impact of changes on security and (2) to reduce the lifespan of security policy violations in the source code by reducing the computing time of taint analysis. The thesis of this dissertation is:

> Driven by changes in source code, we can perform more efficient and faster taint analyses by reducing the parts of the graph explored and, subsequently, use it to find impacted vulnerabilities.

6.1 Summary of Contributions

In the following section, we summarize the main contributions of this project.

This project's most important contribution is the algorithms that incrementally update the taint. We found an original way of propagating the changes from the granularity of the lines of code to the taint in the data-flow graph. By making the algorithms reflect the changes in the source code in the data-flow graph, we significantly reduced the time needed to obtain the taint over the whole application for the systems analyzed. By analyzing over 100 revisions of two open-source projects, we found an average time reduction of 90 to 95 percent compared to the other taint analyses used as a baseline.

We also looked at how the traces could benefit from our incremental taint analysis. We presented some algorithms that generate traces and focused on differential trace generation. By using the results from the incremental taint analysis, we only computed new traces while ignoring already existing traces. This also resulted in significant time reduction.

Another part of the research was to investigate how we can merge together the various sinks and validators APIs into equivalent classes to reduce the number of analyses required. This was motivated by the fact that a different sink may require different validators to ensure the data's integrity. We found that, for the project we analyzed, we could group the sinks together into five different groups.

Finally, we investigated if it is worth using the already computed taint to filter the traversable nodes during the generation of traces. The original idea was that it is useless to go look at

the untainted part of the graphs since they will never lead back to a source. We limited our research to the shortest path generation and found that it slows down the analysis slightly. The overhead of looking at if a node is tainted had a stronger impact on time compared to the paths it filtered.

6.2 Limitations of Proposed Solution

In the following paragraphs, we present the most important limitations of our research.

The main limitation of this project is the lack of projects analyzed. While the number of versions analyzed for every benchmark is sufficient, analyzing Roller and WebGoat is insufficient to conclude if our strategy reduces the analysis time for the average project.

To also obtain better results, we should have run our analysis multiple times on every version of the benchmarks. With more results, we could have mitigated more the variations caused by external factors like the inconstencies in performance of the cloud.

One other problem in our research is the lack of investigation on the formal aspects of our algorithms. We confirmed on the benchmarks analysed that our results obtained with the incremental algorithms are identical to the baseline algorithms. However, we did not formalize that our algorithms will effectively give exactly the same results. We also did not explicitly confirmed the monotonicity of our algorithms and that they are conservative. Future research is needed to confirm the soundness of our algorithms.

We proposed a solution that works well for computing the taint on our analyzed benchmarks. However, further research is needed to look at the portability of our strategy on other static analysis problems. If our algorithms are adapted for other purposes, like finding live variables, we do not know how well it would perform. Our research is strictly limited to problems related to the taint.

6.3 Future Research

In this section, we will look at various improvements that can be made in future research.

Firstly, to address one of our main limitations, one important next step to develop further research is to analyze more projects that are diversified enough to reveal how our analysis performs in a broader manner. Also, we should analyze more languages than Java. It would be interesting to look at how well our analysis performs in other languages and compare the differences. We would like, in particular, to look at languages, like JavaScript or Python, that are not strongly typed, like Java. The way programs are coded in these languages may

be different enough to give different results.

In a future research, the impacts on memory of our incremental strategy should be evaluated. In our work, we only looked at the impacts on time. However, our incremental strategy requires that taint results are either kept in memory or stored on disk. We should evaluate how much more memory does it need and how it scales with the size of the code base.

In future research, we would like to use our incremental strategy to look at other static analysis problems. We can look at not only other similar data-flow problems, like reaching definitions and live variables, but also investigate other subjects, like the call graph generation and the type inference. The goal of trying to answer various problems with our incremental strategy would be to eventually propose a more generic approach. If feasible, this approach could be used as a template for computing various static analysis problems incrementally with granularity at the level of lines of code.

Finally, one improvement that we would like to investigate is how well taint analysis can be integrated in developer tools with our improvements. We would like to integrate the incremental taint analysis in a way that it is constantly updating in real time when the developer updates his or her code base. Such an approach would require that the computation is fast enough not to hinder the developer and show him or her, as fast as possible, sinks that become tainted. With such a tool, surveys of developers could be performed to look at how this affects the developer workflow.

BIBLIOGRAPHY

[1] "Cloc – count lines of code," http://cloc.sourceforge.net/, (Accessed on 04/06/2018).

[2] "Git," https://git-scm.com/, (Accessed on 04/12/2018).

[3] "Github - apache/roller: Mirror of apache roller," https://github.com/apache/roller, (Accessed on 04/06/2018).

[4] "Ibm security appscan source," https://www.ibm.com/marketplace/ibm-appscan-source, (Accessed on 03/27/2018).

[5] "Owasp top 10 - 2017," https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf, (Accessed on 04/08/2018).

[6] "Category:vulnerability - owasp," https://www.owasp.org/index.php/Category:Vulnerability, (Accessed on 04/02/2018).

[7] "Category:owasp webgoat project - owasp," https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, (Accessed on 03/12/2018).

[8] "Allpaths.java," https://algs4.cs.princeton.edu/41graph/AllPaths.java, (Accessed on 02/28/2018).

[9] "T.j. watson libraries for analysis (wala)," http://wala.sourceforge.net/, (Accessed on 03/27/2018).

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

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

[12] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[13] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* ACM, 2016, pp. 332–343.

[14] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill, "Cheetah: just-in-time taint analysis for android apps," in *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on.* IEEE, 2017, pp. 39–42.

[15] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Highly precise taint analysis for android applications," 2013.

[16] R. D. Gordon, "Measuring improvements in program clarity," *IEEE Transactions on Software Engineering*, no. 2, pp. 79–90, 1979.

[17] A. Groce and D. Kroening, "Making the most of bmc counterexamples," *Electronic Notes in Theoretical Computer Science*, vol. 119, no. 2, pp. 67–81, 2005.

[18] W. Le and S. D. Pattison, "Patch verification via multiversion interprocedural control flow graphs," in *Proceedings of the 36th International Conference on Software Engineering.* ACM, 2014, pp. 1047–1058.

[19] T. J. Marlowe and B. G. Ryder, "An efficient hybrid algorithm for incremental data flow analysis," in *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, 1989, pp. 184–196.

[20] R. K. Medicherla and R. Komondoor, "Precision vs. scalability: Context sensitive analysis with prefix approximation," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on.* IEEE, 2015, pp. 281–290.

[21] T. B. Muske, A. Baid, and T. Sanas, "Review efforts reduction by partitioning of static analysis warnings," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on.* IEEE, 2013, pp. 106–115.

[22] U. Nayak and U. H. Rao, *The InfoSec handbook: An introduction to information security.* Apress, 2014.

[23] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, 1995, pp. 49–61.

[24] B. G. Ryder, "Incremental data flow analysis," in *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM, 1983, pp. 167–176.

[25] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[26] M. Sagiv, T. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," *Theoretical Computer Science*, vol. 167, no. 1-2, pp. 131–170, 1996.

[27] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," in *ACM Sigplan Notices*, vol. 44, no. 6. ACM, 2009, pp. 87–97.

[28] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and scalable security analysis of web applications," in *International Conference on Fundamental Approaches to Software Engineering.* Springer, 2013, pp. 210–225.

[29] M. A. Weiss, *Data structures and algorithm analysis in Java.* Pearson, 2011.