Applying Code Analysis and Machine Learning Techniques to Improve Compatibility and Security of Programs

Gábor Antal

Department of Software Engineering
University of Szeged

Szeged, 2021

Supervisor:

Dr. Rudolf Ferenc

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
OF THE UNIVERSITY OF SZEGED

University of Szeged
Ph.D. School in Computer Science

"If you can't fly then run, if you can't run then walk, if you can't walk then crawl, but whatever you do you have to keep moving forward."

— Martin Luther King Jr.

Preface

I clearly remember the first time I used a computer. I can say that it changed my life. Soon after, we bought our first PC and my journey started. My first projects were quite small, like a timetable generator for my classmates, as we took a variety of language and elective classes. My first real project was a management system for an association of junior firefighters; I was 16 years old at that time. As soon as I received the requirements, I started to work on the system as I was very enthusiastic. When I thought I finished the project, the software went live. Soon I got several bug reports, and other feature requests. Of course, my code was a complete mess; it was full of code duplications and security holes (an authorized user accidentally „hacked" it once). Aside from the reported bugs, I discovered several bugs myself that I had to fix (everywhere). Enhancing the software was almost as big of a challenge as writing it from scratch. This was the time I realized that not only are the functional requirements important, the quality and the security of the software are also crucial. I have been working to advance this principle through my doctoral studies ever since. I hope my work in this field helps developers in building better software.

Although the thesis emphasizes the author's contribution, none of the presented research works would have been possible without the help of others. First and foremost, I would like to thank my supervisor, Dr. Rudolf Ferenc, for his guidance and his useful advice that helped me throughout my studies. His positive and calm attitude helped me a lot; without him, I would have probably never done any scientific research. My special thanks go to Dr. Péter Hegedűs, whom I consider my second mentor. He taught me a lot of indispensable things about research, and helped me many times over the years. My sincere thanks go to Dr. Tibor Gyimóthy, the former head of the Department of Software Engineering, for supporting my research work. I would like to express my gratitude to Dr. Csaba Nagy and Dr. Gábor Szőke with whom I started my scientific journey. My many thanks go to my colleagues and article co-authors, namely Dr. Zoltán Tóth, Dávid Havas, Dr. István Siket, Dr. Árpád Beszédes, Dr. József Mihalicza, Márton Keleti, Balázs Mosolygó, Norbert Vándor, Péter Gyimesi, and Dr. Dénes Bán. I would like to thank NNG LLC for providing the interesting topic of backporting C++ code. I wish to thank Edit Szűcs for reviewing and correcting my thesis from a linguistic point of view.

Last, but not least, I wish to express my gratitude to my family for providing a pleasant background conducive to my studies, and also for encouraging me to go on with my research.

Gábor Antal, 2021

Contents

Preface **iii**

1 Introduction **1**
1.1 Structure of the dissertation . . . 3

2 Transforming C++11 Code to C++03 to Support Legacy Compilation Environments **5**
2.1 Overview . . . 5
2.2 Related Work . . . 6
2.3 Approach . . . 7
2.4 Source Code Transformation Framework . . . 10
2.4.1 Source code transformation . . . 10
2.4.2 Incrementality . . . 11
2.4.3 Operation of the Transformation Framework . . . 12
2.4.4 First analysis . . . 12
2.4.5 Tracing the transformed code back to the original one . . . 13
2.5 Transformation Catalog . . . 13
2.5.1 In-class data member initialization . . . 13
2.5.2 Auto type deduction . . . 14
2.5.3 Lambda functions . . . 15
2.5.4 Attributes . . . 16
2.5.5 Final and override modifiers . . . 16
2.5.6 Range-based for loop . . . 16
2.5.7 Constructor delegation . . . 17
2.5.8 Type aliases . . . 18
2.6 Evaluation . . . 18
2.6.1 Functional testing . . . 19
2.6.2 Performance testing . . . 20
2.7 Limitations . . . 23
2.8 Summary . . . 23

3 A Comparative Study on Static JavaScript Call Graph Algorithms **25**
3.1 Overview . . . 25
3.2 Related Work . . . 26
3.3 Approach . . . 27
3.3.1 Overview of the study process . . . 27
3.3.2 Call graph extraction tools . . . 28
3.3.3 Comparison subjects . . . 31
3.3.4 Output format . . . 33

3.3.5 Graph comparison 33
3.3.6 Manual evaluation 34
3.3.7 Performance measurement 35
3.4 Results 35
3.4.1 Quantitative analysis 35
3.4.2 Qualitative analysis 36
3.4.3 Performance analysis 42
3.4.4 Discussion of the results. 43
3.5 Threats to Validity 43
3.6 Summary 44

4 Combining Static and Dynamic Code Analysis with Machine Learning to Detect Software Issues in JavaScript Programs 45
4.1 Overview 45
4.2 Related Work 48
4.2.1 Issue prediction using software metrics 48
4.2.2 Issue prediction using call graphs 49
4.3 Vulnerability Prediction with Static Source Code Metrics Only 52
4.3.1 Approach 52
4.3.2 Results 58
4.4 Enhancing Bug Prediction with Hybrid Call-Graphs 61
4.4.1 Approach 61
4.4.2 Results 66
4.5 Threats to Validity 73
4.6 Summary 74

5 Studying Typical Security Issues and Their Mitigation in Open-Source Projects 77
5.1 Overview 77
5.2 Related Work 79
5.3 Exploring the Security Awareness of the Python and JavaScript Open Source Communities Using The Software Heritage Graph Dataset . . . 80
5.3.1 The Software Heritage Graph Dataset 81
5.3.2 Approach 81
5.3.3 Results 83
5.4 A Data-Mining Based Study of Security Vulnerability Types and their Mitigation in Different Languages 86
5.4.1 Approach 87
5.4.2 Results 89
5.5 Threats to Validity 93
5.6 Summary 95

6 Conclusions 97

Appendices 99

A Summary in English 101

B Magyar nyelvű összefoglaló **107**

Bibliography **115**

List of Tables

1.1 Mapping of thesis points and chapters 4

2.1 Possible transformation scenarios and their fitness (1 - bad, 5 - good) from different angles. 9
2.2 Properties of the subject systems 19
2.3 Code snippets for functional testing 20
2.4 Detailed runtime data without parallelization (in seconds) 21

3.1 Comparison of the used tools (as of 16th July, 2018) 29
3.2 Selected Node.js modules for test 32
3.3 Characteristics of the random generated JavaScript files 32
3.4 SunSpider results 34
3.5 Basic statistics gathered from Node.js results 36
3.6 Precision and recall measures for tools and their combinations 41
3.7 Performance measurements (memory in megabytes, runtime in seconds) 42

4.1 Calculated static source code metrics 55
4.2 F-measures achieved by the machine learning algorithms 59
4.3 Descriptive statistics of the HNII and HNOI metrics calculated using different thresholds 66
4.4 Top 10 recall measures 67
4.5 Top 10 precision measures 68
4.6 Top 10 F-measures 70
4.7 Wilcoxon signed-rank test results of F-measures between models using different feature sets 72
4.8 The best results of the nine ML models according to their F-measure . 72

5.1 Commit statistics per year 84
5.2 Most referenced CWE categories and their description 85
5.3 Average total code changes 92
5.4 Most common CWEs per languages 93

A.1 Thesis contributions and supporting publications 106

B.1 A tézispontokhoz kapcsolódó publikációk 112

List of Figures

2.1 General use case of the framework 9
2.2 Flow chart of the transformation framework 11
2.3 In-class member initialization examples 14
2.4 Auto type deduction examples 15
2.5 Lambda function example 15
2.6 Attribute examples 16
2.7 Final and override modifier examples 16
2.8 Range-based for loop example 17
2.9 Constructor delegation example 17
2.10 Type alias examples 18
2.11 Results of parallel runs (runtime in seconds) 21
2.12 Average distribution of time spent in transformation phases 22

3.1 Methodology overview 28
3.2 Venn diagram of the true/total number of edges found by the tools . . 37

4.1 Data processing overview 52
4.2 Results on the imbalanced dataset 58
4.3 Impact of re-sampling on the learning precision and recall 60
4.4 The schematic view of our approach 62
4.5 Hybrid call graph framework architecture 63
4.6 Venn diagram of found edges 64
4.7 The best Deep Neural Network (DNN) configurations for all three feature sets 68
4.8 The best Deep Neural Network (DNN) configurations for all three feature sets 68
4.9 The best Support Vector Machine (SVM) configurations for all three feature sets 69
4.10 The best Support Vector Machine (SVM) configurations for all three feature sets 69
4.11 The best Random Forest (Forest) configurations for all three feature sets 70
4.12 The best Random Forest (Forest) configurations for all three feature sets 71
4.13 The best K-Nearest Neighbors (KNN) configurations for all three feature sets 71
4.14 The best K-Nearest Neighbors (KNN) configurations for all three feature sets 72

5.1 The schematic view of our approach 81
5.2 Vulnerability mitigation ratio per year 83

5.3 Number of security issues found with the given CWE types 84
5.4 Average number of days between mitigation commit date and CVE publish date grouped by years 85
5.5 Average number of days between commit date and issue publish date for the most common CWEs 86
5.6 A schematic representation of our mining process 87
5.7 The average time elapsed between finding and fixing a CVE (in days) . 89
5.8 The average time elapsed between the publication and fixing of a CVE (in days) 90
5.9 The correlation between the base score (severity) and time taken fixing CVEs 91
5.10 The average number of contributors between the finding and fixing commit 91
5.11 The average number of commits between the finding and fixing of a CVE 92
5.12 Most commons CWEs for Ruby, BitBake, Scheme 94

Listings

2.1 Content of a `compile_commands.json` file 10

3.1 Example of our unified JSON output 33
3.2 A false call edge found by npm-cg 38
3.3 A true call edge found by ACG 38
3.4 A true recursive call edge found by Closure 39
3.5 A confusing code part from string-unpack-code.js 39
3.6 A true call edge found by WALA and TAJS 40

4.1 Example diff file 54
4.2 Example JavaScript function 54
4.3 Sample output from the HNII, HNOI Counter 65

To my beloved family...

1 Introduction

Nowadays, practically everything is powered by software. Years ago, we could not imagine that there will be a day when even our toothbrushes are connected to our phones, providing feedback on how well we brush our teeth. This day has already come quite a few years ago. Everyone uses software, whether for daily work or leisure. Because of this, the amount of software being built is increasing. In 2020, more than 60 million new repositories were created on only GitHub[1] (for comparison, GitHub reached 100 million repositories in November 2018[2]). Building new software is hard enough, but we cannot forget about the difficulties in maintaining existing ones either. Moreover, cyber-crime activities are rising as well, since the possible attack surface is also growing.

After the outbreak of the new coronavirus, the number of daily cyber-attacks grew by 300%, causing more than 4,000 attacks a day [72]. Cyber-criminals are mercilessly targeting the whole populace [82, 87]; they have struck hospitals, researchers, and people working from home alike. The Brno University Hospital was hit by a cyber-attack in March 2020 [118], so they had to cancel surgeries, shut down the whole IT infrastructure, and suspend all of their coronavirus testing activities. Since the Brno University Hospital was one of the biggest COVID-19 testing laboratories at that time in the Czech Republic, this incident caused severe damage. Another example is the Zoom bombing exploit [156]. Because of many people working from home, numerous schools and companies use the video calling software Zoom on a daily basis to hold classes, and meetings. The vulnerability allowed attackers to intercept authentication and join any conversation they wanted. Since then, many companies banned Zoom either temporarily or permanently [36].

These examples show that we must pay attention to software quality and safety. However, as we are all human beings, we occasionally make mistakes in our code that might go unnoticed for a long time. Ideally, we have enough time to develop quality software. We have the freedom to choose what version of which programming

---

[1] Reported on `https://octoverse.github.com/`, a captured version is available on `http://web.archive.org/web/20210330122502/https://octoverse.github.com/`

[2] According to GitHub's blog: `https://github.blog/2018-11-08-100m-repos/`

language we would like to use (of course, we would mostly choose the newest version of a language), we can thoroughly review all frameworks and libraries, and choose the perfect one (with the nicest developer community), and we have plenty of time to do code reviews and cross-validation of each other's code.

Unfortunately, this ideal state hardly exists in real life. The customer might already have a list of constraints with the exact version of a programming language, the available libraries, and tools. However, developers like to learn new things, and they want to stay up to date with the latest technologies. This is not only a natural need, using newer language versions also helps with writing more effective, more expressive, and less error-prone code (e.g. Java 1.7 introduced the AutoCloseable interface along with the try-with-resource construct[3], so that files left open accidentally can be avoided). In order to help developers achieve more freedom in using newer language standards, we studied this field and the current state-of-the-art solutions. We found that there are not any tools that help developers use a newer standard of C++ in such a way that the code becomes „almost" compatible with an older standard of the language automatically. We designed and implemented a complete solution that enables developers to use a large portion of the C++11 language standard in a legacy environment [164]. Our tool is able to convert the source code so that it complies with the C++03 language standard.

However, this is not the only problem developers have to face from time to time. They often have strict deadlines by which time the software must work, meaning that the software has to fulfill the functional requirements, sometimes at the expense of quality and/or prudence. Therefore, developers often use quick solutions that seem to work at first glance, but they do not consider all the possibilities that might happen at some point; nevertheless, such a seemingly innocent line of code can later cause serious (security) issues. Additionally, they often use third-party components from the open-source world. Due to time pressure, they might select the first library/framework that looks suitable, even though there might be a better component that is safer to use, has better quality, has better test coverage, and so on. After the deadline, of course there is another deadline, so developers do not have time to return to a previous (seemingly fixed) issue and reconsider the choices they made. These kinds of problems are even more strongly present in dynamically typed languages, like JavaScript and Python. As there is no compiling phase, there are no static type checks. Additionally, the use of reflection is common in these languages, so that even a human might not easily understand what happens in the code. Thus using static and dynamic code analysis techniques can be fruitful in this context. However, the results of static analyzers are not precise due to the above-mentioned reasons. Many code analysis tools rely on the call graph representation of the program: they produce a call graph to calculate metrics on (for example, the number of outgoing invocations (NOI)). The call graph is the basis for many other, more complex data structures (such as the control flow graph), which is essential in order to detect issues in a given program. As call graphs are fundamental data structures in many ways, the precision of call graphs is extremely important in order to be reliable to the code analysis algorithms. However, there are several tools and algorithms for call graph construction that can be used. Being able to determine which one to use in a given situation is essential. We performed a comparative study on the most popular state-of-the-art static JavaScript call graph construction algorithms,

[3]`https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html`

and presented our findings [165]. We found that there are tools that are somewhat better than others, but we couldn't declare an absolute winner. Based on this study, we integrated ACG.js into the OpenStaticAnalyzer[4] tool, which is developed by the Department of Software Engineering of the University of Szeged.

Using source code metrics for predicting software issues is quite a mature technique [132, 135, 34, 119]. However, the field of prediction for dynamic languages is rather new, and they mostly suggest files that might be vulnerable. If there was a prediction model that works at method level (or on a more fine-grained level, such as line level), developers could use it to prevent issues in their (modified) code. But the practical adoption of the prediction models depends on their real-world performance and the level of false-positive hits they produce. First, we created a fine-grained JavaScript vulnerability dataset that contains the static analysis results of 12,125 JavaScript functions with indicators for whether they contain a vulnerability or not. We also presented a comprehensive comparison of 8 well-known machine learning algorithms on predicting vulnerable JavaScript functions [169]. Our preliminary results were fairly great using only static source metrics, so we extended our prediction model with dynamic analysis and widened our scope from vulnerability to generic bugs. Our model performances improved by replacing static invocation metrics with their counterparts coming from static and dynamic analysis (i.e. hybrid analysis) [168]. But keeping both static and hybrid metrics yields the best results.

Using code analysis tools and prediction models are great automated ways to help spot issues in the code before changes take effect in the software's live version. Nevertheless, the human factor is also not negligible. Bugs will most likely occur in the code, to which we cannot be prepared enough, but we can take a deeper look at security issues. Vulnerabilities in the codebase also happen from time to time; understanding them helps us enhance our prediction models further. It also helps to emphasize how to avoid these defects on the source code when we write educational materials. Knowing the typical security issues might also help developers to know what to look for in a source code when reviewing a change. We used the Software Heritage Graph Dataset to mine data, in order to help JavaScript and Python developers learn their languages' typical security issue types and their characteristics [166]. We also defined an approach on how to mine data for this purpose (from any Git repository). We provided a toolset, and we presented our findings on the typical security issues in several languages [167].

1.1 Structure of the dissertation

The thesis is structured as follows. First, Chapter 1 provides a short introduction to the main work presented in this thesis. The following four chapters are the thesis points, briefly summarized in Table 1.1. In Chapter 2, we present our findings on supporting legacy compilation environments in C++11. We introduce our approach to code transformation, what language features we support, how we transform the source code, and how we support incremental, iterative analysis. We also present our test suite on which we tested our framework, and how we made the tool's performance better. Chapter 3 presents a comparative study on static JavaScript call graph building algorithms. We describe our approach, our testing methodology, as well as the call graph building algorithms. We assess the tools both qualitatively and quantitatively,

[4] https://openstaticanalyzer.github.io/

we calculate the tools' precision and recall values, along with their F-Measure. We conclude the chapter with our findings on which tool is the most advantageous in certain situations. In Chapter 4, we present our prediction model on predicting issues in JavaScript functions. We provide a new dataset (using static source code metrics) on which we trained 8 well-known machine learning algorithms to predict vulnerable functions. We used these algorithms to predict bugs in JavaScript functions. We extended our feature set with two dynamic invocation metrics that helped our model be more accurate. In Chapter 5, we present our findings on typical security issues and their mitigation in several languages, based on the Software Heritage Graph Dataset, and on our own dataset. We describe our data collecting process, we introduce our open-source tools that can help anyone create studies like this. We present our findings on several, widely used programming languages, we present the typical security issue types, and how fast given communities react to them.

In Chapter 6, we sum up the thesis and suggest some directions for further research. At the end, appendices A and B contain a brief summary of the presented thesis in English and Hungarian. In addition, the appendices contain the thesis points, and the contributions of the author, as well as the underlying publications.

Table 1.1. Mapping of thesis points and chapters

№	**Thesis point**	**Chapter**
I.	Transforming C++11 Code to C++03 to Support Legacy Compilation Environments	Chapter 2
II.	A Comparative Study on Static JavaScript Call Graph Algorithms	Chapter 3
III.	Combining Static and Dynamic Code Analysis with Machine Learning to Detect Software Issues in JavaScript Programs	Chapter 4
IV.	Studying Typical Security Issues and Their Mitigation in Open-Source Projects	Chapter 5

2

Transforming C++11 Code to C++03 to Support Legacy Compilation Environments

2.1 Overview

Today, technologies used in software engineering practice, such as programming languages, environments, and libraries, are changing at an inexperienced pace. And, naturally, developers would like to exploit the advantages of such developments in order to increase their productivity, quality of code, and to reduce risks of error. However, there are often certain constraints in the projects that prohibit the use of the newest technologies. This includes, for instance, interoperability with legacy systems, compatibility with older hardware and software, and other limitations arising from the context of the project. For instance, in a situation when the software vendor delivers software to a customer, it must conform to the customer's requirements regarding platform compatibility.

The work presented in this chapter was motivated exactly by such a situation. NNG LLC, one of our industrial partners, is a company that develops navigation software, and as such, it delivers software products to its clients who integrate the navigation software component into the host system of the final product. These host systems often raise strict technical constraints against the delivered software to be integrated. Compatibility may be required with old operating systems, libraries, and existing components. Consequently, the development company needs to enforce strict regulations in-house regarding the usable platforms, language versions, and development environments. The net effect is that the developers are confronted with a situation in which they are limited by older technologies, while they would be eager to use more advanced ones. Often, this leads to lower productivity and even a lack of motivation because their professional skill development is limited as well.

In this thesis point, we present work dealing with the above-mentioned problems in the context of the C++ language, the primary technology used by the company. For many years, the official language standard was not updated, until 2011, which progressively resulted in the birth of a large codebase globally, which is now treated

already as legacy code. The C++11 standard [74] included so many new features (such as in-class initializations, lambda functions, automatic types, attributes, and many more) that almost made it a new language (even Bjarne Stroustrup, the creator of C++ thinks it "feels like a new language[1]"). However, even several years after the publication of the new standard, developers at NNG are still forced to use older versions of the language, which is a significant drawback from both the subject systems and the developers' point of view.

Hence, the goal of our R&D collaboration project was to develop a solution to this problem in a way that would be both beneficial for the developers and the system itself. We created a source code transformation framework with which C++ source code written according to the C++11 standard can be automatically "backported" to C++ code conforming to earlier language versions (C++03, in particular [73]). The framework is capable of automatically transform a large number of new language constructs to their equivalent versions in the older language. This way, developers are free to exploit the latest language features, while production code is still built by using a restricted set of available language constructs. Even though various technical limitations prevented us from making a complete transformation solution in terms of supported language elements, our framework enables a very large subset of C++11, making it usable in practice.

The transformation framework includes a number of additional features besides transforming individual source code files, which make its integration into practical build processes easier. These include, among others, source tree mirroring, incremental transformation, selective transformation, and traceability between the original and the transformed code. The technology has been experimentally integrated into the development process of the company (which was not trivial due to some unique properties of the build process), enabling them to benefit from using recent technology while retaining compatibility with their partners using legacy systems.

This chapter reports on the technical details of the transformation engine, and our experiences in applying it not only on NNG's codebase, but on another industrial application, and on four open source systems as well. Although the transformations do not cover C++11 in 100%, our results and experiences with industrial systems indicated that in its present state, the framework is definitely useful in practice. The transformation engine is available open-source:
`https://github.com/sed-szeged/cppbackport`

The chapter is organized as follows. Section 2.3 presents more details on the practical scenario that lead to the development of the solution. Related work is briefly presented in Section 2.2. Section 2.4 describes the framework and its usage scenarios in detail, while the transformations themselves are listed in Section 2.5. Section 2.6 deals with the evaluation of the solution and our measurement results, together with Section 2.7, which lists the most important limitations of the approach, before the summary in Section 2.8.

2.2 Related Work

This chapter deals with static code analysis for the purpose of source-to-source code transformation. The topic has a large literature, and there are many experimental

[1] `http://www.stroustrup.com/C++11FAQ.html#think`

and production tools developed for various languages, both free and commercial. Also, the application areas are diverse: language translation, (back)porting, modernization, refactoring, etc. In this section, we overview the common solutions for source transformation with a special focus on the C++ language, and not particularly on the application of transformation.

Legacy systems written in languages like Cobol, Fortran, or even C and C++ are often the subject of source transformation, turning them into more modern languages like Java [31, 150, 114, 147].

Compiler infrastructures are often used for language translation, for instance the EDG front end [48], GNU GCC [62], the ROSE compiler infrastructure [138], and LLVM clang [95], which is the chosen platform for our tool as well.

There are solutions that not only offer a library for source transformation, but a complete framework for this task. These frameworks often provide their own language to define the transformation. They are easier to use because they are specific to a particular field. Though they often bring higher overhead, are harder to learn and provide less flexibility. For example, Lee et al. [90] created such an environment. It is highly flexible, and can be extended with new languages as well. A similar system was offered by Bagge et al. [22]. It provides support for source code instrumentation and optimization transformations, but their system only supports C++. There are additional experimental and commercial systems that could possibly be suitable for similar tasks, such as SrcML [41], TXL [152], ASF+SDF [21], Stratego [148], DMS toolkit [47], and several others.

We found that only LLVM provides a proper interface to its internal representation that is suitable for our purposes, so we are using this environment. A few additional applications based on the LLVM clang [95] front end are listed below. Clang Tools [40] is a toolset that includes a code transformation module as well. An interesting tool is modernizer, which transforms C++03 code to C++11; exactly the opposite of what we developed. This tool is appropriate for other tasks as well, such as formatting and code style checking. Another application of this library is Include What You Use [71], with which the optimization of include files can be performed.

Transformation on C++ code for a different purpose was done by, for instance, Aigner et al. [16]. Their software can be used to eliminate virtual function calls in C++ in order to improve the performance of the programs. Marangoni et al. [100] implemented a tool with which general C++ code can be automatically transformed to CUDA source code, which enables parallel execution of general C++ on video cards. Additional parallelization transformation tools have been implemented by Krzikalla et al. [84] and Magni et al. [98].

An interesting tool based on LLVM is C Backend [94], which is able to transform C++ code to C code. This could potentially have also been a solution to our problem (as most compilers still support C), however, this system is still in a very experimental phase. The generated code is much slower than the original, furthermore, it cannot handle a number of code constructs at all.

2.3 Approach

iGO navigation software, the core product of NNG, is a *white-label* product, meaning that clients can sell the final products under their own brand. Clients have significant freedom in customizing the user interface and application behavior to their taste, which

produces high variability not only on the market, but on a technical level as well. While customizations have a big impact on certain features and workflows, many core functionalities remain practically the same in the majority of the products. As a typical software product line [129], the iGO system has core assets that share a common codebase, which has to compile in all supported environments.

In some segments, successful products have numerous new generations with newer and newer versions of the iGO core in them, but without significant changes in the hardware/OS layers. iGO core assets are required to support compilation environments for these legacy platforms as long as business interest [26] and support periods sustain the need. Two notable examples of such legacy target platforms are Windows CE and QNX 6.5. Windows CE can only be targeted with C++03 compilers, while for QNX 6.5 the compilation toolchain is based on GCC 4.4.2.

On the one hand, we see a clearly articulated C++03 compatibility requirement for several years. On the other hand, C++11 and the more recent versions of the C++ language are not only minor refinements, but contain significant benefits over the legacy language. There are multiple aspects here. One group of them relates to product quality. Move semantics of C++11 allows faster code even without modifying the source code [108]. Many features of the new language help to enhance code expressiveness. Self-explanatory code without boilerplates is less error-prone, which in turn leads to better quality and faster production.

The other key factor is developer retention/attraction. Not having major changes to C++98, in a few years, we can refer to C++03 as a 20-years technology. Continuous learning is a vital part of the successful developer mindset [102]. Reliable extension and maintenance of a multiplatform C++ software product line require skilled engineers, for whom modern C++ is the norm. Being forced to use a 20-years technology with millions of lines of code in a non-trivial domain easily becomes a business issue because of this human factor.

The opposing business needs for the legacy and new C++ variants gave NNG the idea of building a bridge between the two. The requirement is simple: the ability to use as many of the modern C++ features in the common codebase as possible without compromising compatibility with the still important legacy platforms.

Our first cooperation in this topic was a classic research project to come up with possible approaches and their detailed assessment for decision making. Table 2.1 contains the identified scenarios and their fitness from different angles. The three possibilities were: *Columbus*, a C++ analysis framework developed at the University of Szeged [55], the open-source *clang* front end for the LLVM infrastructure [95], and the *C backend* developed also for LLVM [94]. Each criterion was assessed on a scale of 1–5, as can be seen in the table. Finally, NNG decided to choose the *clang* code transformation approach, mostly because it is open-source while Columbus is not, and the C backend turned out to be incomplete and unreliable.

A high-level overview of the transformation process is depicted in Figure 2.1. Developers use a modern C++ IDE (e.g. Microsoft Visual Studio[2] 2015) in their daily work. Our tool generates the backported equivalent of the source tree, so when a legacy build or debugging is needed, legacy tools/IDEs (e.g. Microsoft Visual Studio 2005 or GCC 4.4.2) can be used naturally.

Apart from the transformation itself, our framework provides support for various every day software engineering activities, such as testing and debugging. Since run-

[2]https://www.visualstudio.com

Table 2.1. Possible transformation scenarios and their fitness (1 - bad, 5 - good) from different angles.

Criterion	**LLVM clang**	**Columbus**	**LLVM C backend**
Cost of development	2	2	1
Cost of integration into NNG processes	4	4	3
Learning curve	5	5	3
Degradation of work efficiency	3	3	1
Diagnostics	4	4	1
Performance: *compilation*	2	1	1
Performance: *speed*	4	4	1
Performance: *memory*	4	4	3
Performance: *executable size*	4	4	3
New language elements	1	1	4
Robustness	3	3	5
Future proof	1	1	3
Automation	5	5	5
Impact on iGO code	5	5	5
Support	3	4	1
Legacy compatibility	5	5	5

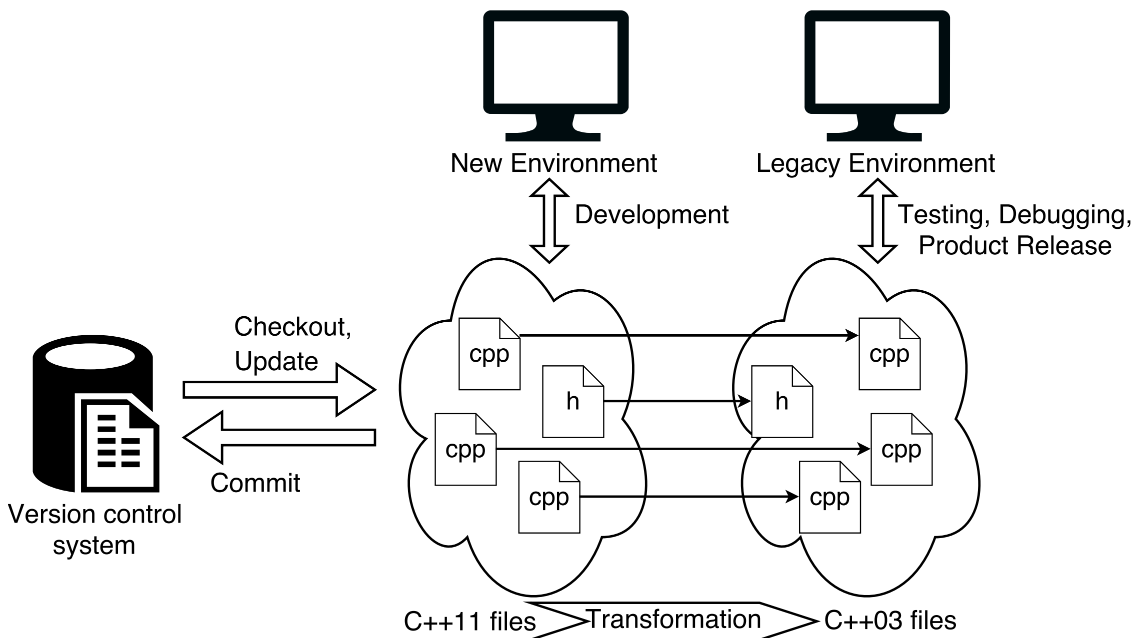


Figure 2.1. General use case of the framework

time issues (either from testing or operation phases) arise at the legacy production environment, while the developers should use their native development environment, the necessary traceability needs to be established on the source code level.

For instance, bug reports of native systems may contain location references to the compiled executable. In case of a crash, for example, call stacks of different threads are dumped. This information together with a corresponding map file that matches the

raw addresses to the source code are invaluable for finding the root cause of the bug. On legacy targets call stacks refer to the backported source code. For more seamless integration into the development processes, we have created a convenience tool that enables developers to look up the source code location in the modern C++ source code even for addresses referring to the backported executable.

2.4 Source Code Transformation Framework

The alteration of the source code is controlled by the transformation framework. It consists of two main parts: the first one is the engine providing incrementality, while the second one is responsible for performing the actual transformations. The incrementality engine monitors the code changes at file level and determines which files of the project need to be transformed (discussed in more detail in Section 2.4.2). Based on this list, the transformation engine performs the necessary changes, which is the topic of Section 2.4.1.

During the design of the framework, it was an important requirement that the tool could be easily integrated into the build processes; either as a pre-build step in traditional build systems or as a subtask in continuous integration (CI) environments.

2.4.1 Source code transformation

For using the transformation framework, we have to know how the compilation units are compiled in their original build environment. We use the `compile_commands.json` file [79] for this purpose (this file is also used by *clang*). This text file contains the necessary information, which is the following:

- **directory**: the working directory used during the build process. The following fields (command, file) are relative to this path.
- **command**: the command line used to compile the compilation unit.
- **file**: path of the compilation unit file.

This data has to be provided for each compilation unit. In Figure 2.1 we show an example `compile_commands.json` file content. If the project does not contain this file yet, then the user has to create it. The `compile_commands.json` file can be created automatically (with an external tool like CMake) or manually.

We did not prepare such a tool on our own, because the industrial partner did not require it and there are several working tools.

Listing 2.1. Content of a `compile_commands.json` file

```
1 [{
2       "directory": "c:/work/projectDir",
3       "command": "cl.exe -c Source1.cpp -o2",
4       "file": "c:/work/projectDir/Source1.cpp"
5 }]
```

Before the transformation starts, the framework copies the full project hierarchy into a work directory, which has to be provided by the user. The transformed code will be saved into this directory as well, so this code will be compilable with a C++03 compiler.

During designing the process it was important to take into consideration that there are also new C++11 features that cannot be transformed in one step (e.g. lambda expressions nested into other lambdas), and some transformations depend on each other and have to be performed in more iterations in a predefined sequence.

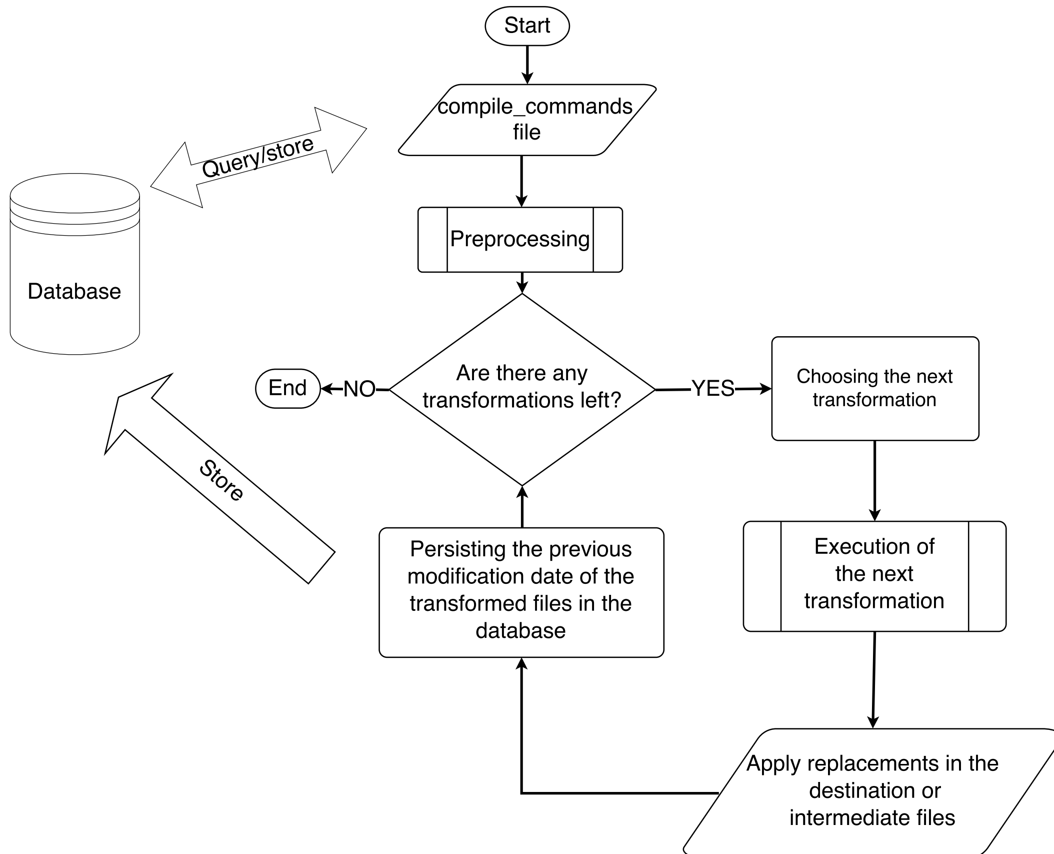


Figure 2.2. Flow chart of the transformation framework

The transformation process and its phases are shown in Figure 2.2. These phases are the following:

- The transformation tool expects the `compile_commands.json` file containing the project's compilation information as input.
- We maintain a database, which supports the incremental operation by storing the latest modification times and the dependencies between the source elements. During preprocessing, the transformation framework analyzes the dependencies between compilation units and selects those files which have to be transformed based on the database (see Section 2.4.2).
- It then iterates over the list of transformations. (We will describe these in Section 2.5).
- After a transformation is done on all affected files, the framework saves the changes, and the incrementality engine updates the database with the file modification dates.

2.4.2 Incrementality

It would take a lot of resources to transform every file during each build of the project. This would be superfluous in most cases, because usually only a small fraction of the

code gets changed during a development iteration. To eliminate this overhead, for each compilation unit file the framework records which version of it was already transformed, and it performs the transformation only if the file was modified in the intervening time. A file is considered to be modified if its last modification time changed. This is not a perfect solution, as the time attribute of a file can change even if its content does not, but this happens quite rarely and the side effect is not harmful.

Because we need to preserve the information between consecutive runs of the framework, we store the data in a persistent storage. We chose the SQLite [3] SQL-engine, because it does not need a database server and can be used easily without any configuration. However, the framework can be quickly adapted to other SQL engines (e.g. PostgreSQL, MySQL), if needed.

The database stores information about the compilation units (which are defined in the `compile_commands.json` file) and associated files for each unit. For each compilation unit, it stores the last modification date, the file dependencies (such as due to inclusion or given in command line arguments), and the timestamp of the dependency addition. If a translation unit includes a header file, which also contains includes, these dependencies will be added directly to the compilation unit, rather than to a dependency. (Dependent files cannot have dependencies this way.) Taking this into account, we developed the following simple database schema:

```
COMPILATION_UNIT(id, timestamp, cmd_args)
FILES(id, path)
RELATIONS(file_id, dep_id, dependency_timestamp)
```

2.4.3 Operation of the Transformation Framework

The framework collects the compilation units from the `compile_commands.json` file and by iterating over this list it also collects their dependencies (direct and indirect ones as well). After this step, the framework compares this information with the contents of the database. If a compilation unit

- changed,
- its command line arguments changed,
- its dependencies changed,
- new dependency appeared, or
- existing dependency disappeared,

then the compilation unit gets inserted into the list of files to be transformed together with its dependencies.

This list contains all files which might have gotten modified since the last transformation, thus the framework will perform the transformation of these files. If all transformations finish successfully, the framework updates the database by saving the new modification dates, adding possible new dependencies, or deleting the disappearing ones. Furthermore, if new compilation units were added, these will also be inserted into the database together with their dependencies.

2.4.4 First analysis

Before starting the first analysis, the framework creates the database. If it already exists, it will not be overwritten. Next, the data tables will be created (if needed).

[3]https://www.sqlite.org/

During the first run, the framework will transform all files, which can be time-consuming in the case of a larger project. Later, however, because of the incrementality, only the changed files will be transformed.

2.4.5 Tracing the transformed code back to the original one

The traceability tool is a complementary tool for the transformation framework, which aims to create a mapping between the transformed and the original project that is able to trace the lines between the original and the converted files. This is useful in cases where the transformed code contains an error, which, of course, has to be fixed in the original code. If it receives a transformed file and the line number in question, it returns the corresponding line in the original file. The tool has been developed in such a way that it can be also called from C code.

Using the tool is limited in the sense that the back tracing can only be performed if it does not fall into a transformed region of code. If a line inside a transformed code part has been selected, it returns the back trace of the starting line of the outermost transformation. The reason for this restriction is that in case of some transformations the body of the transformed functions has to be written out with a procedure provided by clang. The problem is that while this code will be functionally the same as the original, it will differ in formatting. Perhaps the simplest example is that comments and blank lines are not printed out.

2.5 Transformation Catalog

In this section, we present the transformation details of the actual language elements supported by the framework. There are some other transformations available as well, which are in an experimental phase and are mentioned in Section 2.7.

NNG's selection of which language elements to transform was based on their subjective usefulness/benefit judgement and the required efforts and complexity.

2.5.1 In-class data member initialization

The possibility to initialize class (union, struct) data members directly within their declaration in the class body has been introduced in C++11. This has the benefit that a data member which has a default value need not be initialized in each constructor, except only once, directly after its declaration. Earlier, this was only possible for data members with the `const static` modifiers. The syntax for this construct is to use the assignment operator or the brace initializer of the form `{ value }`. The construct has a restriction which is that only one member of unions can be initialized this way.

The listing in Figure 2.3 shows examples for in-class member initialization. The left-hand side of the figure lists the original C++11 code, and the other is the transformed version (C++03). The mechanism used for the transformation is practically the same as the one used by the compiler. Namely, we move data initializers into the constructors, provided they are not already present in the constructor initialization lists. Automatically generated constructors need special consideration. If they are not already generated by the front end, then our transformation framework will create them with public access specification (placed after the last existing member declaration in order not to accidentally modify the visibility of other members).

```
struct A {
  int a { 3 };
  std::string s = "s";


};

union B {
  double a = 3.5;
  int b;

};

class C {
public:
  C(int _b) : b(_b) {
  }
private:
  int a = 1;
  int b = 2;
};
```

⇒

```
struct A {
  int a;
  std::string s;
public: A() : a(3),
              s("s") {}
};

union B {
  double a;
  int b;
public: B() : a(3.5) {}
};

class C {
public:
  C(int _b) : b(_b), a(1) {
  }
private:
  int a;
  int b;
};
```

Figure 2.3. In-class member initialization examples

Some member types are not handled by the framework because they cannot be transformed into their equivalent (or it is not practical). This includes C-style arrays, because their members cannot be directly initialized in the constructor initializer lists, only in the constructor bodies by individual value assignments. Also, declarations in which multiple declarators are provided for the same type are not handled. Finally, code is not transformed for template classes, because in this case there might be constructors which are not instantiated by the front end, so consequently, they could not be used to hold the generated code.

2.5.2 Auto type deduction

Prior to C++11, each variable (and other, entity like a function return value) had to be explicitly declared for its static type. In many cases, this led to overly complex and unreadable code. The `auto` keyword used in place of a concrete type instructs the compiler to deduce the type of the entity automatically. However, in this case, the variable needs to be initialized at the declaration in order for the type to be deducible.

Our transformation framework uses the same deduction rules as the compiler, but in our case, the source code (with the deduced types) is generated as well. In our implementation, various categories of auto types are distinguished, which is necessary because different treatments are required for the different cases:

- simple declarations
- multiple variables in one declaration
- function pointers
- template functions with such variables
- functions with trailing return types

```
1  auto a = 32;
2  auto *b = new auto(&a);
3  auto xp = &a, yp = xp;
4  auto *y = &a, **z = &y;
5  auto foo(int a)
6        -> decltype(a) {
7    return a;
8  }
9  auto x = foo(0);
10 const auto & y = foo(1);
11 auto fp = foo;
```

⇒

```
1  int a = 32;
2  int **b = new int *(&a);
3  int * xp = &a, * yp = xp;
4  int * y = &a, ** z = &y;
5  int foo(int a) {
6
7    return a;
8  }
9  int x = foo(0);
10 const int & y = foo(1);
11 int (*fp)(int) = foo;
```

Figure 2.4. Auto type deduction examples

The listing in Figure 2.4 shows examples for auto type deductions with original and transformed code versions. This transformation has some limitations too. Namely, multiple variables for a declaration in global scope, template functions, and certain variable declarations combined with preprocessor macros are not fully handled.

2.5.3 Lambda functions

One of the most advanced new features in C++11 is the lambda function. With them, special functionalities may be written inline in a very compact way, without actually creating new functions each time, which was only possible using function pointers or function objects in previous editions of C++. Our transformation engine translates lambda functions to function objects, as shown in the example in Figure 2.5.

```
1  std::vector<int> v(6);
2  int inc = 7;
3
4
5
6
7
8
9
10 std::for_each(
11   v.begin(),
12   v.end(),
13   [&inc](int &n) {
14     n += inc;
15   }
16 );
```

⇒

```
1  std::vector<int> v(6);
2  int inc = 7;
3  class LambdaFunctor__12_1{
4    int& inc;
5  public:
6    LambdaFunctor__12_1(
7      int& inc) : inc(inc) {}
8    void operator()(int & n){
9      n += inc;
10   }
11 };
12 std::for_each()
13   v.begin(),
14   v.end(),
15   (LambdaFunctor__12_1(inc))
16 );
```

Figure 2.5. Lambda function example

2.5.4 Attributes

The reason for the introduction of attributes in C++11 was to unify the creation of various compiler directives. Most compilers already implemented their dialect-specific ways for such directives, but this was not standard in any way (for example, construct like `__attribute__((...))` for GNU GCC and `__declspec()` for Microsoft's compiler). The use of attributes makes these kinds of extensions more portable, furthermore, they are very general and might be placed virtually at any syntactic position in the code; they might be placed in namespaces, can get parameters, etc.

```
1 [[attr1, attr2, attr3(args)]]
2 [[namespace::attr(args)]]
```

Figure 2.6. Attribute examples

Figure 2.6 shows what kind of attributes are accepted by our transformation framework. Since in the previous language versions there are no equivalent or similar code structures, we simply discard any occurrence of attributes from the code.

2.5.5 Final and override modifiers

The final and override modifiers were introduced to give developers compile-time control over class specialization and function overriding. These modifiers are not keywords in the language, and depending on the environment they can also appear as e.g. variable names. The `override` modifier indicates that the virtual function of the base class is being overridden. The `final` modifier can be used with both virtual functions and classes. In case of a function, it prohibits its overriding, while in case of a class, it disables subclassing. The framework simply deletes these modifiers, similarly as in the case of attributes. The listing in Figure 2.7 shows examples and their transformed versions.

```
1 class A {
2   virtual void b();
3   virtual void c() final;
4 };
5 class B final : public A {
6   void b() override final;
7 };
```

⇒

```
1 class A {
2   virtual void b();
3   virtual void c();
4 };
5 class B: public A {
6   void b();
7 };
```

Figure 2.7. Final and override modifier examples

2.5.6 Range-based for loop

In order to use the `for` loop easier in cases where an operation has to be performed on a whole range of elements, a more compact way of writing code was introduced. If the given container object has all the required special functions, it can be used in this simplified form. These special functions are called `begin` and `end`. An exception from

this requirement are simple arrays, because in this case the range can be determined by calculating memory address offset. The special functions can be global or local. They are local if the two methods are defined in the class declaration and have no parameters, and global if they are defined outside the class in its enclosing namespace and have a parameter of the required class type.

```
1 int array[4]={1,2,3,0};
2
3
4 for (auto &k : array) {
5
6
7   k = 1;
8 }
```

⇒

```
1 int array[4]={1,2,3,0};
2 int * __begin1 = (array);
3 int * __end1 = (array)+4;
4 for(;__begin1 != __end1;
5          ++__begin1) {
6   int &k = *__begin1;
7   k = 1;
8 }
```

Figure 2.8. Range-based for loop example

During the transformation, the new compact syntax is converted to the old form with three arguments, as shown in the listings in Figure 2.8. We would like to note that the introduced local variables are suffixed with a number to avoid name collision with further transformations in the same scope.

2.5.7 Constructor delegation

C++11 allows the delegation of constructors. This means that in the constructor initialization list another constructor can be called. In this case, the constructor initialization list can only contain this single element. By using constructor delegation a great amount of copied code can be avoided when several constructors would perform similar initializations.

```
1  class A {
2    A() {}
3    A(string str) : s(str)
4    {
5      t = "hello";
6    }
7    A(string str, int dbl)
8      : A(str) {
9
10     a = dbl;
11   }
12   int a = 1;
13   string s;
14   string t;
15 };
```

⇒

```
1  class A {
2    A() : a(1) {}
3    A(string str) : s(str),
4                    a(1) {
5      t = "hello";
6    }
7    A(string str, int dbl)
8        : a(1), s(str) {
9      { t = "hello"; }
10     a = dbl;
11   }
12   int a;
13   string s;
14   string t;
15 };
```

Figure 2.9. Constructor delegation example

The framework transforms the code in such a way that it copies the initialization list of the target constructor into the initialization list or body of the caller constructors,

as can be seen in Figure 2.9. If the constructor delegation is used in template classes then the framework can transform only the instantiated constructors.

2.5.8 Type aliases

Supporting typedef-names is a long-standing feature of C and C++ to create aliases for existing types, but it does not support aliases that can receive template parameters. C++11 introduced a new syntax to support this feature with the `using` keyword. Using template parameters can come in handy in the case of creating aliases for template classes. The listing in Figure 2.10 shows an example.

The framework converts the new syntax into the old format in simple non-template cases in a straightforward way. When there are template parameters, it creates a struct carrying the alias name and it inserts a typedef with the name 'type' into it. Also, all references to the alias are replaced by this construct. The listing in Figure 2.10 shows the transformed example code. Occurrences of the alias name in symbol import statements (`using` from base class, for example), and dependent names as alias parameters (requiring `typename` prefix for the nested type) are currently not supported.

```
1  using ul = unsigned long;
2  ul foo(ul p) {return p;}
3
4  template<class T>
5  using mapVec=std::map
6    <T, Vec<T> >;
7
8
9
10 mapVec<int>
11 bar(mapVec<int> p) {
12   return p;
13 }
```

⇒

```
1  typedef unsigned long ul;
2  ul foo(ul p) {return p;}
3
4  template<class T>
5  struct mapVec {
6    typedef std::map
7      <T, Vec<T> > type;
8  };
9
10 mapVec<int>::type
11 bar(mapVec<int>::type p) {
12   return p;
13 }
```

Figure 2.10. Type alias examples

2.6 Evaluation

We evaluated our transformation framework from two aspects: correctness of the transformed code and performance (runtime). The first aspect is clearly important since we want the transformed code to be functionally equivalent to the original. However, we note that there are language constructs that are not handled by the framework, so these were excluded from our measurements. We discuss functional testing in Section 2.6.1.

The second aspect of the evaluation, performance testing, is important since the framework is planned to be used in production by our industrial partner, integrated into the build process. Since the company employs frequent builds, which is resource intensive due to the large and complex codebase, the time it takes to perform the transformation is also critical. Associated measurements are provided in Section 2.6.2.

During development and the early stages of the evaluation, we used a set of code snippets with the language features of interest. Later we relied on a benchmark of

systems, which use some of the C++11 features, and are non-trivial in size. We included two kinds of systems: four open-source systems and two proprietary ones. Some basic properties of the subject systems are provided in Table 2.2. All subjects belong to different domains, and the sizes of the open-source systems range from small to medium, while the industrial ones can be treated as large systems. The first industrial system is Columbus, our own source code analysis framework [55]. The other system is iGO, the product of our industrial partner, which was the initial motivation for this work.

Table 2.2. Properties of the subject systems

Name	LOC	Translation units	Transformations
SoDA[4]	18,849	126	193
log4cplus[5]	37,543	67	172
GridDB[6]	113,270	68	13
aria2[7]	118,063	385	3,388
Columbus	889,725	1,462	343
iGO	millions[8]	121	0

Lines of Code (LOC), given in the second column is counted as logical lines (not including empty and comment lines), while the number of translation units is essentially the number of source files with the extension `.cpp`, that are compiled by the compiler during build. The last column of the table shows the number of transformations performed by the system during the whole process. It can be observed from the statistics that the actual number of transformed language elements varied from program to program and it did neither really correlate with program size nor with the number of translation units.

The reason behind the surprisingly low number of compilation units in the iGO system is a build time optimization technique called *unity build*. It works by processing a set of compilation units together so that the multiple redundant processing of header files is radically reduced [109]. For iGO, there were no actual transformations performed, which is discussed in the following section.

2.6.1 Functional testing

The correctness of the transformed code was checked in two steps. First, the transformation framework is capable of checking if the code is syntactically correct, so after each successful transformation this check was performed. Second, the code has to produce the same behavior as the original one, and this property was verified at multiple levels:

1. We wrote a set of code snippets containing examples of the implemented transformations (see Table 2.3 for their amount). These pieces of test code have been transformed, syntactically checked, and compiled in the legacy environment.

[4]https://github.com/sed-szeged/soda
[5]https://github.com/log4cplus/log4cplus
[6]https://github.com/griddb/griddb_nosql
[7]https://github.com/aria2/aria2
[8]the exact figure is confidential

Then, each example was manually verified, and finally executed on one or two test cases for functional equivalence. These tests are part of the transformation framework, available open-source.

2. On the four open-source systems and Columbus we also performed the transformation, syntax check, and legacy compilation. Finally, we manually verified a limited number of transformations performed in these systems (due to their large number we could not check all).

3. In the case of iGO, there were no actual transformations performed, as can be observed from Table 2.2 as well. This is because at the time of the experiments the codebase did not include any C++11 features. However, the other parts of the process – analysis, compilation, integration into the build process, incrementality, etc. – were verified. To check the actual working of the transformation engine, the code was temporarily modified at a few places to include C++11 code.

Despite the fact that no actual transformation has been done on iGO yet, the above functional testing process ensures future usability of the framework on this system as well. The transformed code needed to be platform independent, so additionally we performed the tests on both Windows and Linux environments with different compiler versions.

Table 2.3. Code snippets for functional testing

Transformation	**Code snippets**
In-class data member initialization	3
Auto type deduction	37
Lambda functions	31
Attributes	3
Final and override modifiers	3
Range-based for loop	9
Constructor delegation	2
Type aliases	3
All	91

2.6.2 Performance testing

In order to improve the applicability of our framework on big systems we implemented different speedup techniques to reduce the overall processing time:

- Transformation is running on multiple threads in *parallel*.
- *Incremental* transformation only performs the necessary steps based on what has changed since the last transformation.
- *Feature finder* identifies what language features are used in the different compilation units to eliminate their superfluous processing in the unrelated transformation rounds.
- The *MultipleTransforms* phase performs transformations of certain independent language features in a single round.

The following discussion presents measured processing times and other empirical results on our reference codebases.

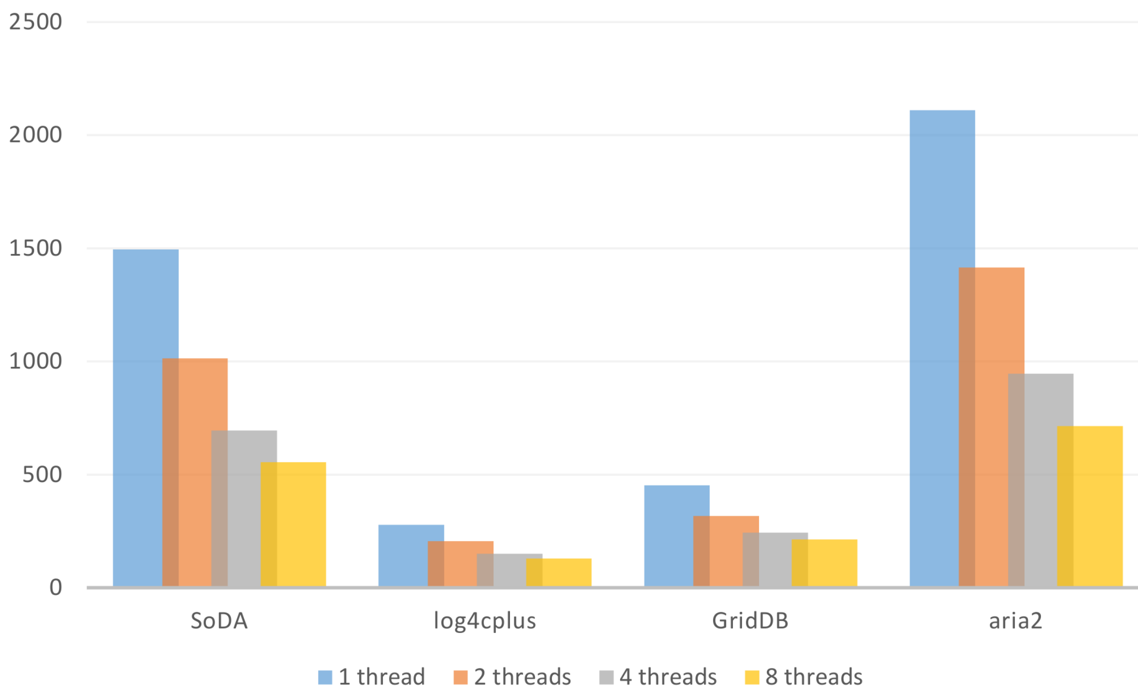


Figure 2.11. Results of parallel runs (runtime in seconds)

Figure 2.11 shows the total processing times on the codebases of the four open-source systems in seconds.[9] The same performance test was performed with different parallelization settings (how many threads to use) to determine the scalability of the framework. Note that *GridDB* has a big advantage in terms of translation time compared to *SoDA*, even though the LOC measure of the former is 6 times that of the latter. The big difference is caused by not including 3rd party code when counting LOC, while the transformation framework has to analyze 3rd party code as well. Systems may have certain large 3rd party codes embedded into their own codebase, resulting in numerous extra instructions that need to be processed by the transformation framework. Currently, the last phase, when syntax check is performed, does not support parallel execution, which reduces scalability to multiple cores.

Table 2.4. Detailed runtime data without parallelization (in seconds)

Name	Dependency Analysis	Feature Finder	Replace Lambda	Multiple Transforms	Remove Auto Delegation	Syntax Check	Total
SoDA	47	853	3	281	35	239	1,458
log4cplus	3	136	3	69	12	48	271
GridDB	4	285	0	57	0	103	449
aria2	20	860	16	508	222	333	1,959
Columbus	142	4,631	73	2,672	617	1,345	9,480
iGO	202	2,319	N/A	N/A	N/A	905	3,426

[9] Source code was accessed via a mapped network drive, presumably resulting in slower than usual file access times, somewhat distorting the measurements.

Table 2.4 presents processing times by phases without parallelization.[10] The transformation starts with *Dependency analysis*, which checks each compilation unit and its dependencies, and decides whether the compilation unit has to be transformed or not. The most time-consuming phase is clearly shown to be *FeatureFinder*, being responsible for identifying language feature usages, because it has to examine all compilation units. The transformation phases (*ReplaceLambda*, *MultipleTransform*, and *RemoveAutoDelegation*) and *Syntax check* phase (which verifies the transformed code) only deal with units containing code fragments relevant to the actual transformation phase. The big differences between the times of *FeatureFinder* and the certain transformation phases reveal how much time is saved by the feature finder optimization. Though transformation phases do not only parse source code, but also transform it, time spent in actual code transformations was measured to be negligible compared to parsing time.

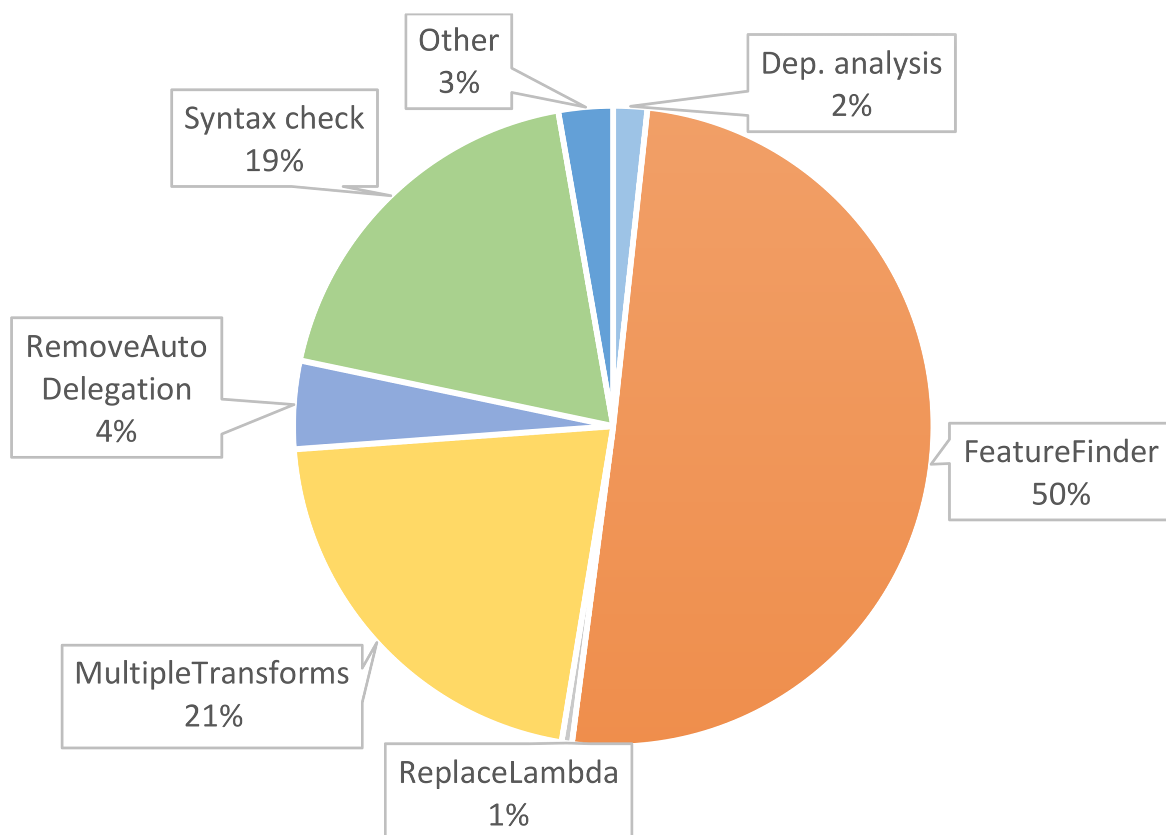


Figure 2.12. Average distribution of time spent in transformation phases

The distribution of the four open-source systems' processing time among the different transformation phases is shown in Figure 2.12. The numbers were determined by averaging the values of Table 2.4. Most of the time (71% on average) is spent in the *FeatureFinder* and *MultipleTransforms* phases. Without *FeatureFinder* the distribution would probably be more equalized, since each phase would contain very similar parsing and negligible code transformation steps for the same complete set of compilation units. *MultipleTransforms* eliminates entire transformation phases by uniting the processing of independent language features.

[10]Although iGO did not contain any C++11 features to transform, the other phases of the process were executed.

2.7 Limitations

Apart from the ones listed in Section 2.5, our framework implements several other transformations, though with limited functionality. This includes the following language features: *variadic templates*, *rvalue references*, *move constructors*, and `decltype` *specifiers* used for type deduction. These features can be used provided some constraints are met by the developers, but since the most typical usage scenarios are handled, this does not mean serious limitations in practice.

In Section 2.5, we already listed some concrete limitations for the transformations (e.g., unused template methods, deletion of attributes). Apart from these, if the framework encounters some specific variants of language features that are not fully handled, it tries to skip those parts and continue the analysis, before eventually terminating with an error. If the system contains code that is generated during compilation, the framework will not consider these files.

Fully automatic generation of the `compile_commands.json` file required for building with the clang infrastructure is not supported. In Linux, the CMake[11] system provides functionality for generating this file, while on other systems Bear[12] might be used. However, some additional modifications are needed to be made on the generated file in order for it to be compatible with the transformation framework. As far as we know, for Windows systems, there is no universal solution for producing the build file, so in this case, the user has to provide it. A particular issue on Windows is related to older Visual Studio versions,[13] in which case the project file has to be prepared (or updated) in multiple versions, one for each Visual Studio edition.

Finally, each subject system to be transformed needs to be compilable by the clang compiler, because this is what our framework is built on. Systems not satisfying this property might require significant porting effort before being capable of transformation.

2.8 Summary

There are many reasons why companies are facing problems when they need to produce C++03 code but their developers are eager to use the new features of C++11. This motivated our work to construct a system for automatically transforming C++11 code to C++03. The system allows, under certain restrictions, for developers to use various C++11 language elements, while after the transformation the software will nevertheless be compatible with the older C++03 standard. We drew a parallel between our framework and other similar systems. We designed the system in a way that it can be easily integrated into a wide range of development processes. In addition, it provides several other services, such as incremental transformation, source code structure cloning, and traceability between the original and the transformed source code.

We detailed the features and capabilities of our source to source transformation system, which includes the basic structure and operation of the framework, the implemented transformations with examples, and information on how we tested them. We evaluated the system's performance on different open-source applications and on two

[11]https://cmake.org
[12]https://github.com/rizsotto/Bear
[13]https://msdn.microsoft.com/en-us/library/ms950416.aspx

large industrial systems, highlighting the scalability and some limitations we encountered. We know that the testing methodology we used for validation could be enhanced further, but current experience shows that the method is already usable in practice.

The developed framework is open-source and it can be used freely. There are many opportunities for further development, however. For instance, handling new language elements, correcting current transformation errors, and improving error recovery mechanisms.

3

A Comparative Study on Static JavaScript Call Graph Algorithms

3.1 Overview

According to GitHub statistics [6] JavaScript is one of the fastest rising languages in years, and it seems that it will continue to dominate in the following years. It had the most pull requests in 2017 and 2016 (according to GitHub's public projects). Each year, the TIOBE Index selects the fastest growing programming language and distinguishes it with the "Language of the Year" award. In 2014, JavaScript was the winner of this award.

Due to its increasing popularity, a lot of projects use JavaScript as their core programming language for both server and client-side modules. Therefore, static code analysis of JavaScript programs became a very important topic as well. Many of the code analysis tools rely on the call graph representation of the program. A call graph contains nodes that represent the functions of the program and the edges between nodes if there exists at least one function call between the corresponding functions. With the help of this program representation various quality and security issues can be detected in JavaScript programs, for example, it can be used to detect functions that are never called or as a visual representation which makes understanding the code easier. We can use call graphs to examine whether the correct number of arguments is passed to function calls or as a basis for further analysis, for example, a full interprocedural control flow graph (ICFG) can be built upon the call graph. With the help of the control flow graphs, various type analysis algorithms can be performed [76, 54, 97, 130]. What is more, this program representation is useful in other areas of research as well, for example, in mutation testing [110], automated refactoring [53], or defect prediction [28].

Being such fundamental data structures, the precision of call graphs determines the precision of the code analysis algorithms that rely on them. Creating precise call graphs for JavaScript, which is an inherently dynamic, type-free, and asynchronous language, is quite a big challenge. Static approaches have the obvious disadvantage of missing dynamic call edges coming from the non-trivial usages of *eval*(), *bind*(), or

apply() (i.e. reflection). Moreover, they might be too conservative, meaning that they can recognize statically valid edges, which are never realized for any inputs in practice. However, they are fast and efficient compared to dynamic analysis techniques and do not require any testbed for the program under analysis.

Therefore, the state-of-the-art static call graph construction algorithms for JavaScript should not be neglected and we need a deeper understanding of their performance, capabilities, and limitations. In this chapter we present and compare some well-known and widely used static analysis based call graph building approaches. We compare five different tools – npm call graph, IBM WALA, Google Closure Compiler, ACG (Approximate Call Graph), and TAJS (Type Analyzer for JavaScript) – quantitatively, to find out how many different calls are detected by the individual tools. We also compare the results qualitatively, meaning that we match and validate the found call edges and analyze the differences. Lastly, we report runtime and memory usage data to be able to assess the usability of the tools on real-world programs.

We found that there are variances in the number, precision, and type of call edges that individual tools report. However, there were considerably large intersections of the reported edges. Based on a manual evaluation of 348 call edges, we found that ACG had the highest precision, above 99% of the found edges were true calls. At the same time, ACG had the highest recall on the union of all true edges found by the five tools; it found more than 90% of the edges. Nonetheless, three other tools (WALA, Closure, npm call graph) found true positive edges that were missed by all the other tools. TAJS did not find any unique edges, however, it achieved a precision of 98% (i.e. comparable to ACG). We also examined the tools in combination and saw that ACG, Closure, and TAJS together found all the true edges, but they also introduced a lot of false ones; their combined precision was only slightly above 83%.

In terms of running time performances, results heavily depend on the size and complexity of the inputs, but Closure and TAJS excel in this respect. From the perspective of memory consumption, for realistic input sizes ACG and Closure overtopped all the other tools. For very large inputs (i.e. in the range of a million lines of code), only Closure Compiler, TAJS, and ACG were able to perform practically efficient code analysis.

The rest of the chapter is organized as follows. In Section 3.2, we list the related literature and compare our work to them. Section 3.3 describes the tool selection and comparison methodology we applied. In Section 3.4, we present the results of our quantitative, qualitative, and performance analysis of the tools. We list the possible threats to the analysis in Section 3.5 and summarize the chapter in Section 3.6.

3.2 Related Work

Using call graphs for program analysis is a well-established and mature technique. The first papers dealing with call graphs date back to the 1970s [52, 65]. The literature is full of different studies built upon the use of call graphs. Clustering call graphs can have advantages in malware classification [83], they can help localize software faults [49], not to mention their usefulness in debugging [136].

Call graphs can be divided into two subgroups based on the method used to construct them: dynamic [158] and static [116]. Dynamic call graphs are obtained by running the program and collecting runtime information about the interprocedural flow [50]. Techniques such as instrumenting the source code can be used for dynamic

call graph creation [46]. In contrast, in the case of static call graphs, there is no need to run the program, it is produced as a result of the program's static analysis. Different analysis techniques are often combined to obtain a hybrid solution that guarantees a more precise call graph, thus a more precise analysis [51].

With the spread of scripting languages such as Python and JavaScript, the need for analyzing programs written in these languages also increased [54]. However, constructing precise static call graphs for dynamic scripting languages is a very hard task that is not fully solved yet. The *eval()*, *apply()*, and *bind()* constructions of the language make it especially hard to analyze the code statically. However, there are several approaches to construct such static call graphs for JavaScript with varying success [54, 32, 58]. Constructed call graphs are often limited, and none of the studies deal fully with EcmaScript 6 since the standard was released in 2015.

Feldthaus et al. presented an approximation method to construct a call graph [54] through which a scalable JavaScript IDE support could be guaranteed. Madsen et al. focused on the problems induced by libraries used in the project [97]. They used pointer analysis and a novel use analysis to enhance scalability and precision. In our study, we only deal with static call graphs for JavaScript and do not propose a new algorithm, but rather evaluate and compare existing approaches.

In his thesis [45], Dijkstra evaluates various static JavaScript call graph building algorithms. This work is very similar to our comparative study, however, it was published in 2014 and a lot has happened since then in this research area. Moreover, while Dijkstra focused on the evaluation of the various conceptual algorithms implemented by himself in Rascal, our focus is on comparing mature and state-of-the-art tool implementations on these algorithms ready to be applied in practice.

There are also works with the goal to create a framework for comparing call graph construction algorithms [91, 17]. However, these are done for algorithms written in Java and C. Call graphs are often used for preliminary analysis to determine whether an optimization can be done on the code or not. Unfortunately, as they are specific to Java and C languages, we could not use these frameworks as is for comparing JavaScript call graphs.

3.3 Approach

3.3.1 Overview of the study process

Figure 3.1 displays the high-level overview of the external and self-developed software components we used in our comparative study. We run each of the selected tools (Section 3.3.2) on the test input files (Section 3.3.3). As can be seen, we needed to patch some of the tools (marked with \) for various reasons (see Section 3.3.2), but mainly to extract and dump the call graphs built into the memory of the programs (all the modification patches are available in the online appendix package[1]). Next, we collected the produced outputs of the tools and ran our data conversion scripts to transform each call graph to a unified, JSON-based format we defined (Section 3.3.4). The only exception was Closure, where we implemented the call graph extraction to the JSON format right into the patch extracting the inner-built call graph, because there was no public option for outputting it, thus it was easier to dump the data right into

[1] `http://www.inf.u-szeged.hu/~ferenc/papers/StaticJSCallGraphs/`

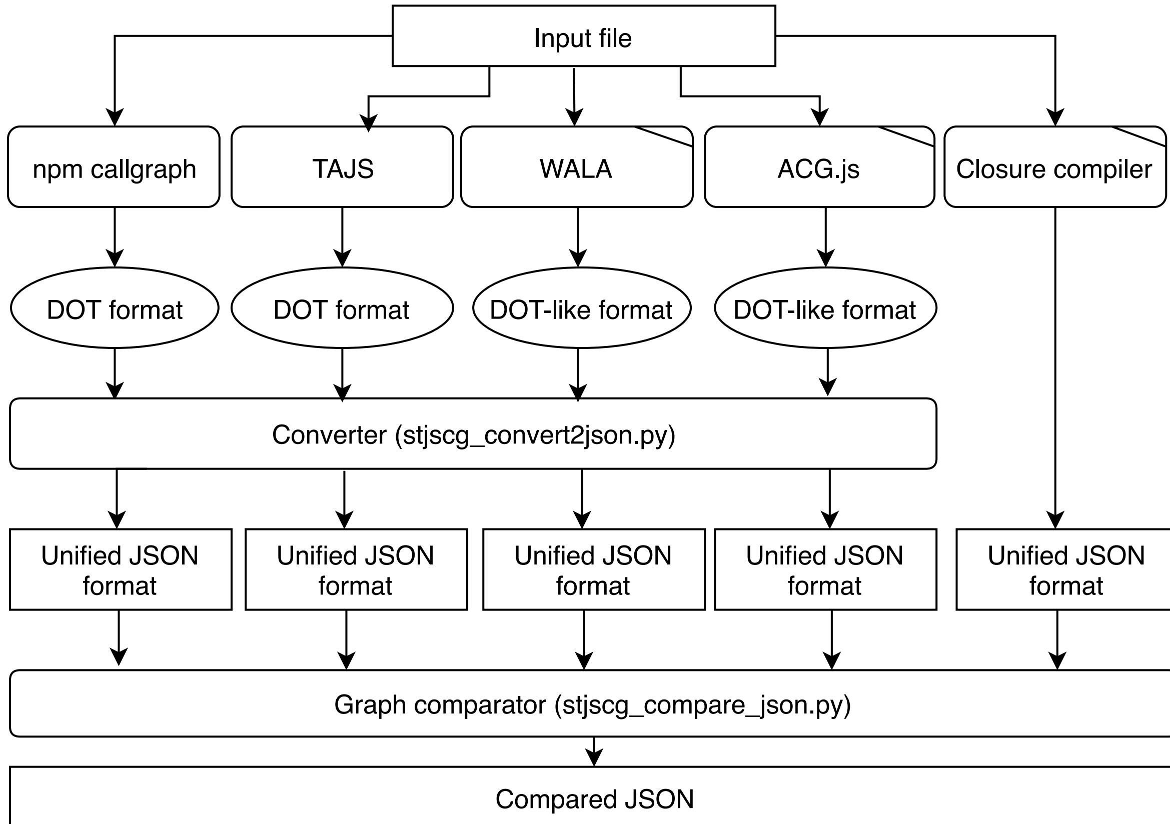


Figure 3.1. Methodology overview

the unified JSON format. In all other cases, we built a custom data parser script that was able to read the output of the tools and produce an equivalent of it in our JSON format. From the individual JSON outputs of the tool results, we created a merged JSON with the same structure using our graph comparison tool (Section 3.3.5). This merged JSON contains all the nodes and edges found by any of the tools, with an added attribute listing all the tool identifiers that found that particular node or edge. We ran our analysis and calculated all the statistics on these individual and merged JSON files (all the produced JSON outputs are part of the online appendix package).

3.3.2 Call graph extraction tools

In this section, we present the tools we took into account in our comparative study. We examined tools that: i) are able to create a function call graph from a JavaScript program, ii) are free and open-source, and iii) are adopted in practice.

It is important to note that in this study we work with call graphs, where:

- Each node represents a function in the program (identified by the file name, line and column number of the function declaration),
- An edge between two nodes is directed and represents a statically possible call from one function to another (i.e. function *f()* may call function *g()*),
- There might only be zero or one edge between two nodes, so we only track if a call is possible from one function to another, but omit its multiplicity (i.e. we do not count at how many call site calls may happen). This is because not all of

the tools are able to find multiple calls and we wanted to stick to the most basic definition of the static call graph anyway.

Based on these criteria, we selected the following five tools for our comparative study (see Table 3.1 for an overview).

Table 3.1. Comparison of the used tools (as of 16th July, 2018)

Tool name	**Language**	**Size (SLOC)**	**Commits**	**Last commit**	**Contri-butors**	**Issues (open/closed)**	ECMAScript **compatibility**
WALA	Java	232,594	5,845	06/11/2018	25	151 (74/77)	ES5
Closure compiler	Java	398,959	12,525	06/16/2018	373	2163 (796/1367)	ES6 (partial)
ACG	JavaScript	120,531	193	10/28/2014	3	7 (1/6)	ES5
npm callgraph	JavaScript	207	30	03/14/2017	2	16 (6/10)	ES6 (partial)
TAJS	Java	53,228	16	01/04/2018	1	10 (6/4)	ES5 (partial)

WALA

WALA [58] is a complete framework for both static and dynamic program analysis for Java. It also has a JavaScript front-end, which is built on Mozilla's Rhino [137] parser. In this study, we only used one of its main features, which is static analysis, call graph construction in particular.

In order to have the output that suits our needs, we had to create a driver which serializes the built call graph. For this, we used an already existing version of the call graph serializer found in the official WALA repository (*CallGraph2JSON.java*). As a first step, we converted the actual call graph to a simple DOT format then we used our converter script to transform this into the final JSON file. WALA produced multiple edges between two functions if there were multiple call sites within the caller function. Since our definition of call graph allows for one edge between two functions in one direction at most, we modified the serializer to filter the edges and merge them if necessary. We had to handle the special case where the call site was in the global scope, as in this case there was no explicit caller method. In such cases, we applied the common practice followed by other tools as well and introduced an artificial "toplevel" node as the source of these edges.

WALA itself is written entirely in Java, its main repository is under active development, mostly by the IBM T.J. Watson Research Center. WALA was used in over 60 publications [13] since 2003.

Closure Compiler

The Closure Compiler [32] is a real JavaScript compiler, which works similarly to other compilers. But in the case of Closure Compiler, it compiles JavaScript into a better JavaScript: it parses and analyzes JavaScript programs, removes dead code, rewrites, and compresses the code. It also checks common JavaScript mistakes.

It builds a call graph data structure for internal use only, which other algorithms (in Closure Compiler) can take advantage of. Therefore, we had to modify the available source code and provide a call graph JSON dump function. Closure Compiler contains the inclusion of the artificial root node by default to represent calls realized from the global scope. The JSON writer filters any duplicate edges (Closure keeps track

of various call sites) to provide an appropriate JSON output that can be used for comparison (see Section 3.3.4).

The Closure Compiler itself is written entirely in Java and is actively developed by Google.

ACG

ACG (Approximate Call Graph) implements a field-based call graph construction algorithm [54] for JavaScript. The call graph constructor algorithm can be run in two basic modes, pessimistic and optimistic, which differ in how interprocedural flows are handled. In our study, we used the default `ONESHOT` pessimistic strategy for call graph construction.

For ACG, we had to implement the introduction of an artificial root edge (i.e. “toplevel”) and filtering of multiple edges, as ACG also tracks and reports edges connected to individual call sites. Moreover, ACG only reported the line numbers of functions in its output, which we had to extend with the column information. All these modifications are available in one single patch.

As there are several forks of the original repository available currently, we had to check all of them and select the one which is the most mature among these forks. The selected one was created by the CWI group from Amsterdam.

The npm callgraph module

Npm callgraph is a small npm module that creates call graphs from JavaScript code developed by Gunar C. Gessner. It uses UglifyJS2 [25] to parse JavaScript code. Despite its small size and few commits, quite a lot of people use it; it has more than 1300 downloads.

TAJS

Type Analyzer for JavaScript [77] is a data flow analysis tool for JavaScript that infers type information and call graphs. It is copyrighted to Aarhus University.

The proposed algorithm is implemented as a Java system that is actively maintained since the publication of the original concept. However, we suspect that this is only an external mirror of an internal repository that is synchronized periodically. It was not necessary to modify the source code of TAJS, as it provides a command line option for dumping call graphs into a DOT format that we were able to parse and convert into our unified JSON format.

Other considered tools

There are of course other candidate tools that could have been involved in this study. We found numerous commercial and/or closed-source programs, like SAP HANA. However, we focused on open-source programs, which are easy to access and can be modified or customized easily to fit our needs.

In our evaluation study, we only dealt with tools directly supporting call graph building either internally or as a public feature. Thus, we were forced to left out some great JavaScript static analysis tools that do not support call graph extraction directly. One such tool was the open-source Flow [5] developed by Facebook, a very popular

static code analysis tool for type checking JavaScript. Unfortunately, Flow does not provide a public API for obtaining the built call graph or a control flow graph. As such, we would have been required to implement our own algorithms above the internal control flow data structure, which would introduce a threat to the validity of this study. Our primary goal in this work was to empirically compare existing call graph extraction algorithms, not to upgrade all tools to achieve call graph extraction.

Other relevant tools we examined were JSAI (JavaScript Abstract Interpreter) [81] and SAFE (Scalable Analysis Framework for EcmaScript) [88], both build an intermediate abstract representation from JavaScript to further perform an analysis on. It is true that they calculate control and data flow structures, but they specifically utilize them for type inference. None of them support the extraction/export of call graphs, hence we were unable to include them in our evaluation study.

The tool `code2flow` [2] also looked like a great choice, but since it is officially abandoned without any follow-up forks, we excluded it from our list. We note that the original repository of ACG was also abandoned, however, it has several active forks on GitHub.

Another reason we dropped possible tools from comparison was immaturity, which means that the given project had one contributor and there was only a very short development period before the project was left abandoned. These tools also lacked documentation, thus their usability was poor. We did not take into account JavaScript Explorer Callgraph [10] due to this reason. Furthermore, we also left out callgraphjs [1] since this project only contains supporting material for ACG.

3.3.3 Comparison subjects

To perform a deep comparison of the tools, we identified three test input groups.

Real-world, single-file examples

The first group consists of real-world, single-file, "bare" JavaScript examples. For this, we chose the SunSpider benchmark of the WebKit browser engine [9], which is extensively used in other works as well. The benchmark programs are created to test the JavaScript engine built into WebKit. Therefore, these programs contain varying complexity code with many different types of functions and calls, but all in one single JavaScript source file. These properties make them an excellent choice for our real-world, single-file test subjects. Moreover, all the programs are of manageable size, thus we could easily check and analyze the calls manually.

Real-world, multi-file Node.js examples

To test the handling of modern, ECMAScript 6 and Node.js features (like module exports or external dependencies, i.e. the *require* keyword) and inter-file dependencies, we collected six popular Node.js modules from GitHub. Our selection criteria included the following: the module should contain multiple JavaScript source files, it should have an extensive test suite with at least 75% code coverage and be used by at least 100 other Node.js modules. The requirements for test coverage comes from our mid-term research goal. We would like to repeat the presented comparative study extended with dynamic call graph extraction algorithms that typically require an existing test suite for

programs under analysis. The details of the chosen Node.js modules are summarized in Table 3.2.

Table 3.2. Selected Node.js modules for test

Program	Repository	Size (SLOC)
debug	`https://github.com/visionmedia/debug`	442
doctrine	`https://github.com/eslint/doctrine`	5,109
express	`https://github.com/expressjs/express`	11,673
jshint	`https://github.com/jshint/jshint`	68,411
passport	`https://github.com/jaredhanson/passport`	6,173
request	`https://github.com/request/request`	9,469

Generated large examples

In order to stress test the selected tools and measure their performances, we needed some really large programs. However, we were unable to find large enough open-source programs that only use those language features that all of the tools recognize. Therefore, we decided to generate JavaScript programs that conform to the ECMAScript 5 standard, as it is the highest standard all the selected tools support.

We defined two categories of such generated inputs. The programs in the *simple* category contain simple function calls with some random statements (variable declarations, object creation, and object property access, for loops, while loops and return statements). They are pretty straightforward without any complex control flows, but their sizes range from moderate to very large. The programs in the *complex* category also contain numerous function calls, but are extended with functions with parameters, callback functions, function expressions, loops, and simple logging statements. These programs are meant to test the performances of tools when parsing complex control flows. The code generation was performed automatically, with custom-made Python scripts.

We generated three files in the simple and two in the complex category (the exact properties of these programs are shown in Table 3.3). After the generation, we used the Esprima Syntax Validator[2] to validate our files, to make sure they are valid JavaScript programs and can be parsed with any ECMAScript 5 compatible JavaScript front-end.

Table 3.3. Characteristics of the random generated JavaScript files

Type	File	Lines of code	Nodes	Edges
Simple	s_small.js	68,741	1,000	49,286
	s_medium.js	382,536	2,600	331,267
	s_large.js	1,321,088	5,000	1,224,251
Complex	c_medium.js	28,544	400	3,000
	c_large.js	413,099	1,000	50,000

[2] `http://esprima.org/demo/validate.html`

3.3.4 Output format

The different selected tools produce their outputs in different formats by default. To solve this problem, we had to process their outputs and convert them into a unified format that can be used for further analysis. We chose a simple JSON format that contains the list of nodes and edges of a call graph demonstrated in Listing 3.1.

Listing 3.1. Example of our unified JSON output

```
{
  "nodes": [
    {
      "id": 1,
      "label": "[simple.js]function1()",
      "pos": "simple.js:10:1"
    },
    {
      "id": 2,
      "label": "[simple.js]function2()",
      "pos": "simple.js:18:1"
    }
  ],
  "links": [
    {
      "target": 2,
      "source": 1,
      "label": "[simple.js]function1() -> [simple.js]function2()"
    }
  ]
}
```

Each node has a unique identifier (*id*), a *label*, and source code position information (we call it *pos*). The position information clearly identifies a function (i.e. node). Each edge connects exactly two of the nodes by their unique ids.

3.3.5 Graph comparison

The quantitative analysis of the call graphs focuses on the comparison of the number of nodes and edges. For the qualitative analysis – inspired by the work of Lhoták et. al [91] –, we created a call graph comparison script written in Python. The script is available in our online appendix package. The aim of the script is to detect the common edges found by different tools. The inputs of the script are the converted unified format JSON outputs (see Figure 3.1 and Listing 3.1) of the tools, and its output is a similar JSON file that contains the union of the found nodes and edges of all tools for each program. The script decorates each node and edge JSON entry with a list of tool identifiers that found the particular node or edge. The identification of nodes and edges is done by precise path, line, and column information, as JavaScript functions have no names, and it would be cumbersome to rely on a unified unique naming scheme anyway. Moreover, the Chrome V8 [11] JavaScript engine follows the same strategy.

To ensure the proper comparison, we manually checked the produced path and line information of the evaluated tools. TAJS reported precise line and column information

in its standard DOT output. In cases of Closure Compiler, WALA, and ACG, we implemented or modified the line information extraction. Unfortunately, WALA was only able to report line numbers, but no column information, thus we manually refined the JSON outputs it produced. In the case of npm callgraph, the reported line information was not precise (neither line, nor column information), thus we went through all the cases manually and added them to the produced JSON files.

Table 3.4. SunSpider results

	npm callgraph		ACG		WALA		Closure Compiler		TAJS	
File	**nodes**	**edges**	**nodes**	**edges**	**nodes**	**edges**	**nodes**	**edges**	**nodes**	**edges**
3d-cube	15	23	15	22	17	24	15	23	15	23
3d-morph	2	1	2	1	0	0	2	1	2	1
3d-raytrace	22	29	28	40	21	22	27	40	28	39
access-binary-trees	3	3	4	3	4	5	4	5	4	5
access-fannkuch	2	1	2	1	3	2	2	1	2	1
access-nbody	8	11	12	15	8	11	11	14	12	15
access-nsieve	3	2	3	2	2	1	3	2	3	2
bitops-3bit-bits-in-byte	2	1	2	1	3	2	2	1	3	2
bitops-bits-in-byte	2	1	2	1	3	2	2	1	3	2
bitops-bitwise-and	0	0	0	0	0	0	0	0	0	0
bitops-nsieve-bits	3	2	3	2	3	2	3	2	3	2
controlflow-recursive	4	6	4	3	4	6	4	6	4	6
crypto-aes	17	16	17	16	13	16	17	16	13	14
crypto-md5	21	30	21	30	3	2	21	30	12	15
crypto-sha1	18	23	18	23	3	2	18	23	9	8
date-format-tofte	18	18	19	20	2	1	3	2	3	2
date-format-xparb	0	0	14	14	13	17	14	14	5	5
math-cordic	5	5	5	5	5	5	5	5	5	5
math-partial-sums	2	1	2	1	2	1	2	1	2	1
math-spectral-norm	6	6	6	6	6	6	6	6	6	6
regexp-dna	0	0	0	0	0	0	0	0	0	0
string-base64	3	2	3	2	3	2	3	2	3	2
string-fasta	5	4	5	4	5	4	5	4	5	4
string-tagcloud	4	4	12	15	2	1	11	17	3	2
string-unpack-code	0	0	5	4	5	8	12	64	5	20
string-validate-input	4	3	5	4	5	4	5	4	5	4
$\sum$	**169**	**192**	**209**	**235**	**135**	**146**	**197**	**284**	**155**	**186**

3.3.6 Manual evaluation

As part of the qualitative analysis of the results, we evaluated all the 348 call edges found by the five tools on the 26 SunSpider benchmark programs. The manual evaluation was performed by two researchers - who participated in this research - going through all the edges in the merged JSON files and looking at the JavaScript sources to identify the validity of those edges. As an output, each edge of the JSON has been extended with a “valid” flag that is either *true* or *false*. After both researchers evaluated the edges, they compared their validation results and resolved those two cases where they disagreed initially. Upon consensus, the final validated JSON has been created.

As for the Node.js modules, the large number of edges made it impossible to validate all of them. In this case, we selected a statistically significant representative random sample of edges (exact numbers can be found in Section 3.4.2) to achieve a 95% confidence level with a 5% margin of error. A researcher - who participated in this research - manually checked all these selected edges in the Node.js sources.

3.3.7 Performance measurement

For our performance measurements, we ran the tools on an average personal computer with Windows 7. The main hardware characteristics were Intel Core i7-3770 processor (at 3.90 GHz), 16 Gb RAM, and 1 Tb HDD (7200 rpm). We note that besides TAJS (which can measure the time of its analysis phases), neither of the tools can measure their own running time and/or memory usage.

To measure the memory usage of the tools uniformly, we implemented a small tool that queries the operating system's memory usage at regular intervals and stores the acquired data for each process. In order to acquire running time data, we modified each tool's source code. For the two Node.js tools (ACG and npm callgraph), we used the `process.hrtime()` method to calculate running time. We also had to set the maximum heap size to 6 Gb for the Node.js based tools.

For the three Java-based tools (WALA, Closure Compiler, and TAJS), we set the maximum heap size to 11 Gb. For running time measurement, we used the timestamps from the `System.nanoTime()` method.

3.4 Results

3.4.1 Quantitative analysis

SunSpider benchmark results.

To evaluate the basic capabilities of the selected tools, we used the SunSpider benchmark for the WebKit browser engine (i.e. the first test program group). This package contains 26 files which we analyzed one at a time with each tool. After the analysis, we collected the different outputs and we converted them to our previously defined JSON format (see Section 3.3.1). We calculated some basic statistics from the gathered data that can be seen in Table 3.4. The table shows the number of nodes (functions) and edges (possible calls between two functions) found by each tool for every benchmark program. As can be seen, there are programs for which the number of nodes and edges are the same for all tools (e.g. bitops-bitwise-and.js, math-partial-sums.js). There are also programs for which the results are very close, but not exactly the same (e.g. bitops-3bit-bits-in-byte.js, string-validate-input.js) and consensus could be made easily. We should note, however, that tools typically produce similar call graphs for small programs with only a few functions, as there is only a small room for disagreement. Finally, there are programs where the numbers show a relatively large variance across the call graph building tools (e.g. 3d-raytrace.js, date-format-tofte.js).

Node.js module results.

To evaluate the practical capabilities of the selected tools, we chose six popular, real-world open-source Node.js modules. Details about the subject programs can be found in Section 3.3.3.

Unfortunately, npm callgraph and WALA were unable to analyze whole, multi-file projects because they cannot resolve calls among different files (e.g., requiring a module). TAJS supports the require command, nonetheless, it was still unable to detect call edges in multi-file Node.js projects. On the other hand, Closure Compiler

and ACG were able to recognize these kinds of calls. Thus, we only used these two tools to perform the analysis on the selected Node.js modules.

Table 3.5. Basic statistics gathered from Node.js results

	ACG		**Closure Compiler**	
Program	**nodes**	**edges**	**nodes**	**edges**
debug	19	15	12	8
doctrine	85	175	53	174
express	82	186	118	239
jshint	349	1001	320	1236
passport	41	40	39	49
request	122	223	123	239
$\sum$	**698**	**1640**	**665**	**1945**

We calculated some basic statistics from the gathered data that is shown in Table 3.5. The table displays the number of nodes (functions) and edges (possible calls between two functions) found by the tools. As can be seen, the results show resemblance; the correlation between nodes and edges found by the two tools is high. However, not surprisingly, there are no exact matches in the number of nodes and edges for such complex input programs. It is interesting though, that for *doctrine* the number of edges found by ACG and Closure Compiler is very close (175 and 174, respectively), but there is a large difference in the number of nodes found by the tools.

3.4.2 Qualitative analysis

For qualitatively comparing the results, we applied our exact line information based call graph comparison tool described in Section 3.3.5. With this, we could identify which call edges were found by the various tools and compare the amount of common edges by all tools, or the edges found only by a subset of the tools.

SunSpider benchmark results.

First, we present the qualitative analysis on the single file SunSpider JavaScript benchmark programs. All the Venn diagrams are available in an interactive version [124] as well in the online appendix package, where one can query the list of edges belonging to each area.

Figure 3.2 presents the Venn diagram of the found call edges in the total of 26 benchmark programs by the five tools. The first numbers show the true edges according to our manual evaluation (see Section 3.3.6), while the second numbers are the number of total edges. The percentages below the two numbers display the ratio of true edges in that area compared to the total number of true edges found by the tools (which is 257 out of 348). This representation highlights the number of edges found by all possible sub-sets of the five tools.

In total, 93 edges were found by all the five subject tools, all of them being true positive calls. However, four of the tools found edges that the others missed. Although WALA, Closure Compiler, and npm callgraph (npm-cg) reported a significant amount

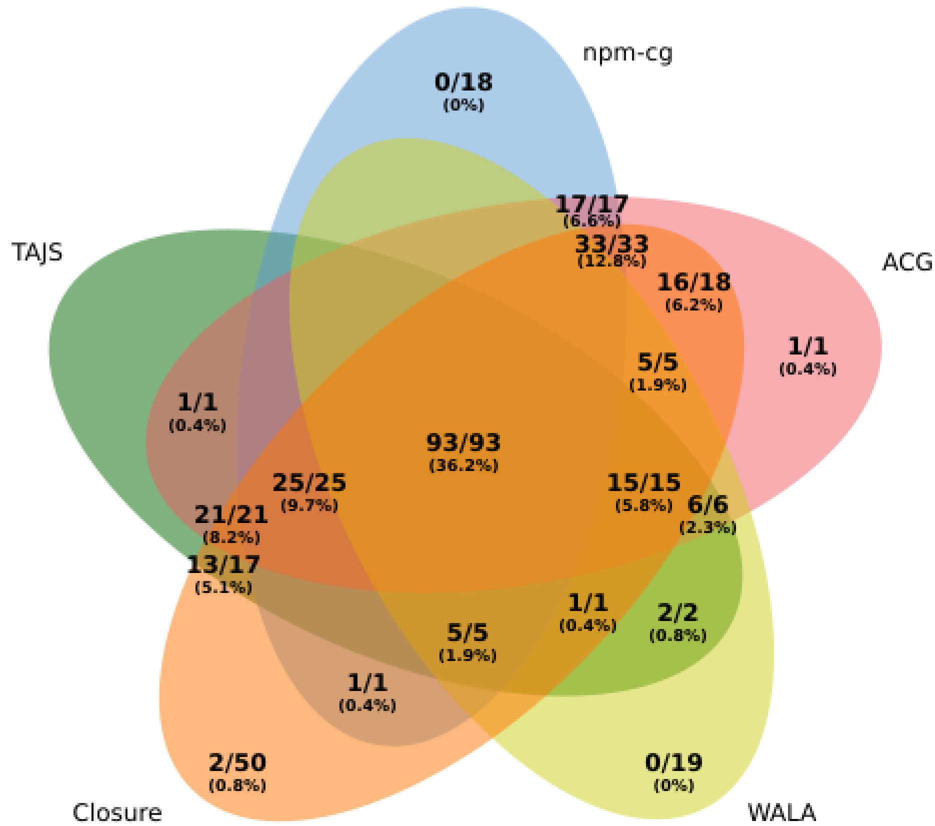


Figure 3.2. Venn diagram of the true/total number of edges found by the tools

of edges that no other tools recognized, most of them turned out to be false positives during their manual evaluation.

Edges found by npm callgraph only The manual analysis of the 18 unique edges found only by the npm callgraph tool showed that all of these are false positive edges. Every edge represents a call from the global scope of the program to a function. Even though the reported calls exist, all of the call sites are within another function and not in the global scope. Listing 3.2 shows a concrete example[3] from the *access-nbody.js* benchmark program. The tool reports a call of `Sun` function (line 1) from the global scope, but it is called within an anonymous function (line 8) from line 10. This call is properly recognized by all the other tools, however.

[3]toplevel:1:1->access-nbody.js:74:1

Listing 3.2. A false call edge found by npm-cg

```
function Sun(){
  return new Body(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, SOLAR_MASS);
}
...
var ret = 0;

for ( var n = 3; n <= 24; n *= 2 ) {
  (function(){
    var bodies = new NBodySystem( Array(
      Sun(),Jupiter(),Saturn(),Uranus(),Neptune()
    ));
    ...
  }
}
```

Edge found by ACG only There is only one edge found by ACG and no one else, which is true positive. It is a call[4] to a function added to the built-in *Date* object via its prototype property in *date-format-tofte.js*. Listing 3.3 shows the excerpt of this call.

Listing 3.3. A true call edge found by ACG

```
Date.prototype.formatDate = function (input,time) {
  ...
  function W() {
    ...
    var prevNY = new Date("December 31 " + (Y()-1) + " 00:00:00");
    return prevNY.formatDate("W");
  }
  ...
}
```

Edges found by WALA only In the case of WALA, all the 19 unique edges are false positive, but for different reasons.

5 of the 19 edges have a target function of "unknown", thus WALA was not able to retrieve the target node of the call edge. We checked these instances manually and found that all these unknown nodes are implied by *Array()* calls. As all the built-ins and external calls are omitted from the analysis, these edges are clearly false ones.

Another group of ten false edges comes from the *date-format-xparb.js* program. This program contains a large switch-case statement that builds up calls to various functions as strings. These dynamically created strings are then executed using the *eval()* command to extend the prototype of the *Date* object with generated formatting functions. These formatting functions are then called from a function named *dateFormat*. WALA recognizes direct edges from *dateFormat* to the functions generated into the body of the formatting functions, which is invalid, as the functions are called from the dynamically created formatting functions that are called by *dateFormat*.

[4]date-format-tofte.js:186:5->date-format-tofte.js:8:29

The last four false edges are due to invalid recursive call edges reported in the *string-unpack-code.js* program. There are several functions identified by the same name in different scopes, but WALA was unable to differentiate between them.

Edges found by Closure only Closure found a couple of recursive edges that no other tool did. For example, Listing 3.4 shows an edge[5] in the *string-tagcloud.js* program, where the *toJSONString* function is called from its body (line 8).

Listing 3.4. A true recursive call edge found by Closure

```
1  Object.prototype.toJSONString = function (w) {
2  ...
3      switch (typeof v) {
4        case 'object':
5          if (v) {
6            if (typeof v.toJSONString === 'function') {
7              a.push(k.toJSONString() + ':' +
8                  v.toJSONString(w));
9          }
```

48 out of the 50 unique edges by Closure is in the *string-unpack-code.js* program. All of them are false positive edges. The reason is that Closure seems to ignore the visibility of identifiers within scopes (similarly to what was observed in the case of WALA). Listing 3.5 shows a sketch of the problematic calls.

Listing 3.5. A confusing code part from string-unpack-code.js

```
1  var decompressedMochiKit = function(p,a,c
2      ,k,e,d){e=function(c){return(c<a?"":
3      e(parseInt(c/a)))+((c=c%a)>35?String.
4      fromCharCode(c+29):c.toString(36))}
5      ...
6  }(...);
7  var decompressedDojo = function(p,a,c
8      ,k,e,d){e=function(c){return(c<a?"":
9      e(parseInt(c/a)))+((c=c%a)>35?String.
10     fromCharCode(c+29):c.toString(36))}
11     ...
12 }(...);
```

The inner function redefining parameter *e* of the outer function (line 2) is called within itself (line 3), which is correctly identified by Closure and TAJS, but no other tool. However, Closure reports edges from the same location to all the other places where a function *e* is called (e.g. line 9), which is false, because that *e* is not the same *e* as it is already in another body block referring to another locally created function denoted by *e*. The *string-unpack-code.js* defines four deeply embedded functions with the same parameter names, hence most of the found edges are false.

Interesting edges found by TAJS TAJS did not find any edges that were missed by all the other tools. However, it did find some interesting edges detected only by one

[5] string-tagcloud.js:99:37->string-tagcloud.js:99:37

other tool. One such call edge is through a complex control flow that was missed by the tools except for TAJS and ACG.

Moreover, TAJS was the only tool besides WALA that detected edges coming from higher-order function calls. Listing 3.6 shows such a call[6] in *bitops-3bit-bits-in-byte.js*.

Listing 3.6. A true call edge found by WALA and TAJS

```
1  function fast3bitlookup(b) {
2    ...
3  }
4  ...
5  function TimeFunc(func) {
6    ...
7    for(var y=0; y<256; y++) sum += func(y);
8    ...
9  }
10 sum = TimeFunc(fast3bitlookup);
```

Validated and combined tool results As we systematically evaluated all 348 found call edges, we could also calculate the well-known information retrieval metrics (precision and recall) for each tool and their arbitrary combinations. We would like to note, however, that evaluation and comparison were done for simple call edges; paths along these edges were not taken into consideration. Missing or extra edges might have different impacts depending on the number of paths that go through them, thus precision and recall values might be different for the found call chain paths.

Table 3.6 contains the detailed statistics of the tools. The first column (Tool(s)) is the name of the tool or combination of tools. The second column (TP) shows the total number of true positive instances found by the appropriate tool or tool combination. In the third column (All), we display the total number of edges found by the appropriate tool or tool combination. The fourth column (TP*) shows the total number of true edges as per our manual evaluation (i.e. it is 257 in each row). The fifth (Prec.), sixth (Rec.*), and seventh (F) columns contain the precision (TP / All), recall (TP* / TP) and F-measure values, respectively.

We must note that Rec.* is not the classical recall measure. We did not strive to discover all possible call edges during manual validation, rather simply checked whether an edge reported by a tool is true or not. Thus we used the union of all true edges found by the five tools as our golden standard. This is just a heuristic, but it provides a good insight into the actual performances of the tools compared to each other.

From the individual tools, ACG stands out with its almost perfect (99%) precision and quite high recall (91%) values. While TAJS and npm-cg maintain similarly high precision (98% and 91%, respectively), their recalls (71% and 68%) are far below ACG's. Closure's recall (89%) is very close to that of ACG, but it has significantly lower precision (81%). WALA has a moderate precision (87%), but the worst recall (49%) in our benchmark test.

Looking at the two tool combinations, ACG+TAJS stand out based on F-measure; together they perform almost perfectly (98% precision and 99% recall). It looks like they complement each other quite well. In fact, they seem to be a perfect combination as there are no other three, four, or five tool combinations that would even come close

[6] bitops-3bit-bits-in-byte.js:28:1->bitops-3bit-bits-in-byte.js:7:1

to this F-measure score. ACG, TAJS, and Closure reach the maximum recall together, while maintaining a precision of 83%. Taking all the tools into consideration, the combined precision decreases to 74% with a perfect recall.

Table 3.6. Precision and recall measures for tools and their combinations

Tool(s)	TP	All	TP*	Prec.	Rec.*	F
npm-cg	174	192	257	91%	68%	77%
ACG	233	235	257	99%	91%	95%
WALA	127	146	257	87%	49%	63%
Closure	230	284	257	81%	89%	85%
TAJS	182	186	257	98%	71%	82%
npm-cg+ACG	239	259	257	92%	93%	93%
npm-cg+WALA	203	219	257	93%	79%	85%
npm-cg+Closure	247	319	257	77%	96%	86%
npm-cg+TAJS	233	255	257	91%	91%	91%
ACG+WALA	241	262	257	92%	94%	93%
ACG+Closure	255	309	257	83%	99%	90%
ACG+TAJS	254	260	257	98%	99%	98%
WALA+Closure	238	311	257	77%	93%	84%
WALA+TAJS	187	210	257	89%	73%	80%
Closure+TAJS	239	293	257	82%	93%	87%
npm-cg+ACG+WALA	242	281	257	86%	94%	90%
npm-cg+ACG+Closure	255	327	257	78%	99%	87%
npm-cg+ACG+TAJS	255	279	257	91%	99%	95%
npm-cg+WALA+Closure	255	346	257	74%	99%	85%
npm-cg+WALA+TAJS	238	258	257	92%	93%	92%
npm-cg+Closure+TAJS	256	328	257	78%	99%	88%
ACG+WALA+Closure	257	330	257	78%	100%	88%
ACG+WALA+TAJS	254	279	257	91%	99%	95%
ACG+Closure+TAJS	257	311	257	83%	100%	90%
WALA+Closure+TAJS	239	312	257	77%	93%	84%
npm-cg+ACG+WALA+Closure	257	348	257	74%	100%	85%
npm-cg+ACG+WALA+TAJS	255	298	257	86%	99%	92%
npm-cg+ACG+TAJS+Closure	257	329	257	78%	100%	88%
npm-cg+TAJS+WALA+Closure	256	347	257	74%	99%	85%
TAJS+ACG+WALA+Closure	257	330	257	78%	100%	88%
ALL	257	348	257	74%	100%	85%

Node.js module results.

As we already described, only ACG and Closure were able to analyze the state-of-the-art Node.js modules listed in Table 3.2. From the 2281 edges found together by the two tools in the six modules, 1304 are common, which is almost 60%. It is quite a high number considering the complexity of Node modules coming from structures like event callbacks, module exports, requires, etc. There were 336 edges (14.7%) found only by ACG and 641 (28.1%) found only by Closure.

As the amount of edges here is an order of magnitude larger than in the case of SunSpider benchmarks, we were not able to entirely validate the found calls manually. However, we evaluated a statistically significant amount of random samples. To achieve a 95% confidence level with a 5% margin of error, we evaluated 179 edges that were uniquely found by ACG, 240 edges from those found only by Closure, and 297 from the common edges. We found that 149 out of 179 (83.24%) edges were true for ACG, 40 out of 240 (16.66%) edges were true for Closure, and 248 out of 297 (83.5%) were true for the common edges.

3.4.3 Performance analysis

In this section, we present the results of the performance testing. We would like to note that the measurement results contain every step of call graph building, including reading the input files and writing the output. That was necessary because different tools implement call graph building in different ways, but reading input and writing output is a common point. We ran the tools ten times on each of the generated inputs and used the averages as a result (see Table 3.7). We highlighted the best runtime and memory consumption in each line.

Table 3.7. Performance measurements (memory in megabytes, runtime in seconds)

		npm callgraph		ACG		WALA		Closure Compiler		TAJS	
Category	**File**	**Memory**	**Runtime**	**Memory**	**Runtime**	**Memory**	**Runtime**	**Memory**	**Runtime**	**Memory**	**Runtime**
Simple	s_small.js	404	13.33	**237**	**3.11**	1151	16.55	519	6.41	718	5.18
	s_medium.js	2234	175.76	**1168**	49.35	2537	181.62	1338	**17.28**	1671	23.83
	s_large.js	5702	1401.88	3338	636.49	8784.22	1085	3277	**50.16**	**3132**	102.91
Complex	c_medium.js	281	4.76	**239**	**2.56**	826	8.27	411	4.92	370	2.74
	c_large.js	3283	76.49	1452	39.63	4010	210.45	**1388**	27.29	2067	**23.79**

In general, Closure, ACG, and TAJS performed best in all cases. The npm callgraph module was generally faster than WALA. But when it comes to large inputs, WALA was 30% faster than npm callgraph. On the other hand, it used more than one and a half times as much memory. The differences may vary with the sizes of the inputs; in some cases, a tool was ~28 times faster (npm callgraph vs. Closure, *s_large.js*), and for another input only ~3% better than the other tool (npm callgraph vs. WALA, *s_medium.js*).

On the medium-sized test set in the complex category, ACG performed the best closely followed by TAJS. On the large set, Closure used the least memory, however, TAJS produced the call graph in the shortest time. It is clearly visible that the more complex problems are considered (more similar to real-world applications) the more variance is present in the runtimes and memory consumptions. We suppose it is due to the different inner representations the tools have to build up in order to obtain a call graph. We assume that Closure and ACG keep their inner representations as simple as possible, consequently, call edges are easily located by them in the case of simple programs. For complex cases, this behavior could be less effective and the more complex inner representations will pay off.

We would like to stress that these results do not say anything about the correctness and accuracy of the produced output, they are simply approximate measurement data of the memory usage and running time performances.

3.4.4 Discussion of the results.

Each approach and tool has its pros and cons. During this comparative study, we distilled the following statements.

- Recursive calls are not handled in every tool; Closure Compiler seems to be the most mature in this respect.
- Edges pointing to inner function (function in a function) declarations are not handled by every tool, e.g. WALA produces a lot of false edges because of this.
- Only WALA and TAJS can detect calls of function arguments (i.e., higher-order functions).[7]
- ACG and TAJS are able to track complex control flows and detect non-trivial call edges.
- Closure often relies on simple name-matching only, which can cause false or missing edges.
- It seems that WALA can analyze *eval()* constructions and dynamically built calls from strings to some extent.
- The calls from anonymous functions defined in the global scope are mistreated by npm-cg, which detects a call edge directly coming from the global scope in such cases.
- Closure has a superior runtime performance for very large inputs with high recall at the expense of lower precision.
- ACG consumes the least amount of memory and runs the fastest among all the tools on small to medium-sized inputs.
- WALA and npm-cg are practically unusable for analyzing code at the scale of millions of code lines.

3.5 Threats to Validity

A lot of factors might have affected our measurements. Some of the tools might perform additional tasks during call graph construction, which we could not omit from the measurement. Nonetheless, we treat our performance measurement with proper care; they are only used to assess the orders of magnitudes for memory consumption and running times.

Our modifications in the tools for call graph extraction mechanism may have introduced some inconsistencies. However, we only made slight changes and most of them only affected the reporting of edges, thus this threat has a limited effect.

We ran all the tools with default configurations. Various parameters might have affected the performance and precision of the tools. Nonetheless, we do not expect the main results to be much affected by these parameters.

[7]We should note, however, that according to its documentation ACG might be able to identify higher-order functions in the optimistic configuration, at the cost of lower precision.

We might have missed some good candidate tools from the comparison. However, the presented evaluation strategy and insights are useful regardless of this. Nevertheless, it is always possible to replicate and extend a comparative study like this.

Regarding the manual evaluation of the call edges, the subjectivity of evaluators is also a threat. We tried to mitigate this by having two participating researchers validate all the edges for the 26 SunSpider benchmark test cases. There were preliminary disagreements in only 2 out of 348 cases between the evaluators that they could resolve in the end. Thus, we think the possible bias due to evaluation errors is negligible.

3.6 Summary

Code analysis of JavaScript programs has gained a large momentum during the past few years. Many algorithms for vulnerability analysis, coding issue detection, or type inference rely on the call graph representation of the underlying program.

We presented a comparative study of five state-of-the-art static algorithms for building JavaScript call graphs on 26 WebKit SunSpider benchmark programs and 6 real-world Node.js modules. Our purpose was not to declare a winner, rather to gain empirical insights to the capabilities and effectiveness of the state-of-the-art static call graph extractors.

Each tool had its strengths and weaknesses. For example, Closure recognized recursive calls and had an overall good performance both in terms of running time and memory consumption, but it introduced errors due to shallow name-matching. It also had a relatively low recall. ACG tracked complex control flows to find call edges and had high precision and recall at the same time with great memory consumption and runtime, but missed higher-order function calls. WALA had the capability to detect higher-order function calls (callbacks), but produced some edges with unknown nodes, and had the lowest recall and highest memory consumption of all tools. The npm callgraph tool had very high precision, but poor performance, and found no unique true call edges. TAJS provided very conservative results, meaning that it had almost perfect precision, but very low recall, while having a very good overall performance.

It is also evident from the results that the combined power of various tools is superior to those of individual call graph extractors. Thus, we would encourage the development of algorithms that combine these state-of-the-art approaches.

Our future plan is to extend and replicate the presented study by adding more static tools (e.g. taking commercial tools and IDEs into consideration) as well as including some dynamic call graph extraction approaches.

4

Combining Static and Dynamic Code Analysis with Machine Learning to Detect Software Issues in JavaScript Programs

4.1 Overview

JavaScript is getting traction not just in client-side web development, but as a desktop and server language (Node.js), mobile app language (React Native), or even as an IoT (e.g. JerryScript or the Espruino framework [4, 60]) implementation language. Therefore, programs written in JavaScript are more and more exposed to possible risks. Regardless of the rapid rise of cyber-crime activities and the growing number of devices threatened by them place software security issues in the spotlight, security concerns of programs are still neglected from time to time. According to past studies, around 90% of all attacks exploit known types of security issues [105].

Due to the dynamic behavior of JavaScript, the source code may not be as easy to understand (and follow) as other, stricter languages. For example, a function with several parameters can be called even without specifying a single parameter. A function without any formal parameters can be called with various parameters, and the function itself can use an object called `arguments`[1] to access the parameters in its body. These and similar constructions exist and are used often in JavaScript programs. Using static code analyzers might help the developers to catch defects in the source code, however, the dynamic nature of JavaScript makes the analysis hard (even building a call graph from JavaScript source code is not unequivocal, as we already discussed in Chapter 3).

On the other hand, we already have a great deal of already fixed issues in the world of JavaScript. Hence involving machine learning might be a good approach to predict possible issues in the early stages; finding vulnerable components for applying existing mitigation techniques on them might be a viable practical approach when it comes to fighting against cyber-crime.

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments`

Issue prediction aims at finding source code elements in a software system that are likely to contain defects. Being aware of the most error-prone parts of the program, one can efficiently allocate the limited amount of testing and code review resources. Therefore, the prediction can support software maintenance and evolution to a great extent. Nevertheless, practical adoption of such prediction models always depends on their real-world performance and the level of annoying misclassification (i.e., false-positive hits) they produce. Despite the relative maturity of the defect prediction research area (in general), the practical utilization of the state-of-the-art models is still very low due to the reasons mentioned above.

Prediction models can use a diverse set of features to build effective prediction models. The most common types of such features are static source code metrics [67, 93, 66, 57], process metrics [70, 144, 96], natural language features [29, 69], and their combination [18, 44, 64]. All these metrics proved to be useful in different contexts, but the performance of these models may vary based on, for example, the language of the project, the composition of the project team, the type of issues one would like to predict, or the domain of the software product. We need further studies to better understand how and when these models work best in certain situations. Additionally, we can refine source code metrics by using static and dynamic analysis in combination, which has a yet unknown impact on the performance of bug prediction models (especially in dynamic languages like JavaScript).

Security vulnerabilities are similar to bugs (i.e. most of them can be seen as special types of bugs, though not functional), however, many studies show that bug prediction models cannot be applied for finding vulnerabilities as is [143, 163]. Although this suggests that specific prediction models are needed for finding vulnerable software components, we can still leverage the abundance of knowledge already accumulated in the area of bug prediction. Moreover, most of the bug prediction models find fault-prone files or classes [78, 120, 141, 162, 112, 39, 163], while rarely working at a finer granularity level (e.g. for methods, functions, or statements [142]). These approaches are less effective in the case of JavaScript, as source code is often structured in only a few files (even into one single `js` file) and usually there are no higher-level logical constructs (like classes) above functions. Prediction models for vulnerable source files would not really be useful in such contexts; we need at least function level vulnerability information and prediction models, so that they can be adopted in practice.

In this chapter, we propose two prediction models using different datasets and different features to predict software issues, namely security vulnerabilities and generic software bugs. We investigate whether predicting defects in functions is feasible based on various software metrics. We compare the performances of the most widely used machine learning algorithms on this prediction task, including two deep neural network variants, the K-Nearest Neighbors algorithm (KNN), a decision tree classifier (Tree), the C-Support Vector Classification variant of the Support Vector Machine algorithm (SVM), Random Forest (Forest), Logistic regression (Logistic), Linear regression (Linear), and the Gaussian Naive Bayes algorithm (Bayes). We apply various re-sampling strategies to handle the imbalanced nature of the dataset.

In Section 4.3, we investigate how the state-of-the-art machine learning techniques perform in predicting functions with possible security vulnerabilities in JavaScript programs. To the best of our knowledge, there are no existing vulnerability datasets for JavaScript programs specifically, which would contain vulnerability information at the level of functions. VulinOSS [61] and VulData7 [78] are very useful proposals with

the aim of collecting general vulnerability datasets together with fixing patches. However, they are not specific to JavaScript and do not map the fixed vulnerabilities to individual functions.

In order to overcome this problem, we created a fine-grained, public JavaScript vulnerability dataset with data extracted from *nsp* (Node Security Platform [7]) and the *Snyk Vulnerability Database* [12] automatically matched with information available on GitHub (i.e fixing commits and patches). The new function level vulnerability dataset contains 12,125 functions from which 1,496 are vulnerable. It includes static code metrics provided by the OpenStaticAnalyzer [8] and escomplex [3] tools as well.

Given the highly dynamic nature of JavaScript, we got encouraging results using only static code metrics as predictors. However, static code analysis may not be very precise due to the dynamism described above. Although static source code metrics proved to be very efficient in bug prediction (in general), the imprecision due to the lack of dynamic information might affect the defect prediction models using them.

In Section 4.4 we propose a function level JavaScript issue prediction model based on static source code metrics with the addition of hybrid (static and dynamic) code analysis based metrics of the number of incoming and outgoing function calls. To support the hybrid code analysis of JavaScript programs, we created a hybrid call graph framework, called *hcg-js-framework*[2] that can extract the call graph information of JavaScript functions using both static and dynamic analysis.

Unfortunately, the database we proposed in Section 4.3 could not be used for this particular task. The vulnerabilities and their fixes came from 93 different programs. In order to gather dynamic traces of the execution from each software, we would have needed a much more complex workflow. Furthermore, we would have needed to ensure that all of the software can be tested with the "npm test" command; to install and manage all of the external dependencies (besides the ones mentioned in `package.json`); to ensure that each and every program has sufficient test coverage. Providing these conditions requires a huge effort. We also would like to apply our machine learning approach to a slightly different, wider field, namely to predict generic bugs. For that, we used a publicly available bug dataset, BugsJS [68]. It consists of 453 real, manually validated bugs and their fixing patches from 10 popular JavaScript projects. From this dataset, we selected the ESLint[3] project to test our hybrid model approach, which contains 333 bugs. Based on the hybrid call graph results of ESLint, which we used as a subject system for bug prediction, we refined the Number of Incoming Invocations (NII) and Number of Outgoing Invocations (NOI) metrics. We added them to a set of common static source code metrics to form the predictor features in a training dataset consisting of 824 buggy and 1943 non-buggy functions extracted from BugsJS. These invocation metrics are typically very imprecise in JavaScript, calculated solely based on static analysis, as a great number of calls happen dynamically, like higher-order function calls, changes in prototypes, or executing the *eval()* function, which is impossible to capture statically. We analyzed the impact of these additional hybrid source code metrics on the function-level bug prediction models trained on this dataset.

We found that using invocation metrics calculated by a hybrid code analysis as bug prediction features consistently improves the performance of the ML prediction models. Depending on the ML algorithm, applied hyper-parameters, and target measure we consider, hybrid invocation metrics bring a 2-10% increase in model performances

[2] `https://github.com/sed-szeged/hcg-js-framework`
[3] `https://eslint.org/`

(i.e., precision, recall, F-measure). Interestingly, even though replacing static NOI and NII metrics with their hybrid counterparts HNOI and HNII in itself improve model performances, keeping them all together yields the best results. It implies that hybrid call metrics indeed add some complementary information to bug prediction.

Our main contribution in this chapter is the publicly released vulnerability dataset consisting of the static analysis results of 12,125 JavaScript functions complemented with the information of whether the functions contain a vulnerability or not. We presented a comprehensive comparison of 8 well-known machine learning algorithms for predicting vulnerable JavaScript functions. We created a hybrid call graph framework, called *hcg-js-framework* in order to support the hybrid analysis of JavaScript programs. We introduced HNII and HNOI metrics and investigated whether the combination of static and dynamic analysis improves the 8 well-known machine learning algorithms on predicting buggy JavaScript functions.

The chapter is organized as follows. In the next section (Section 4.2), we summarize the works that are related to ours. In Section 4.3, we present the approach we used to collect and process data, the used machine learning models, and we present the comparison of the machine learning models with various hyper parameters on predicting vulnerable JavaScript functions. Afterwards, we describe how we collected and mapped ESLint bugs to functions, the extraction of hybrid call graphs, the assembling of the training dataset, and the obtained results in Section 4.4. We enlist the possible threats to our work in Section 4.5. Finally, we summarize the chapter in Section 4.6.

4.2 Related Work

Using source code metrics for predicting software issues is quite a mature technique [132, 135, 34, 119]. Malhotra [99] concluded in his systematic literature review that machine learning techniques have the ability to predict software faults.

4.2.1 Issue prediction using software metrics

In their preliminary study, Siavvas et al. [145] investigated if a relationship exists among software metrics and specific vulnerability types. They used 13 metrics and found that software metrics may not be sufficient indicators of specific vulnerability types, but using novel metrics could help. In our study, we used a lot more (namely 35) static source code metrics, including various Halstead variants, and found that they can effectively predict vulnerable functions in JavaScript.

In their work, Jimenez et al. [78] proposed an extensible framework (VulData7) and dataset of real vulnerabilities, automatically collected from software archives. Although it is similar to our work, VulData7 is general, i.e., it contains vulnerabilities for various languages at the file level. Even though it contains JavaScript vulnerabilities, using them in our study was infeasible, as JavaScript files could contain a lot of functions. The database we presented in this work is more fine-grained; every piece of information is available at the function level, thus enabling more accurate experiments.

Neuhas et al. [120] introduced a new approach (and the corresponding tool) called Vulture, which can predict vulnerable components in the source code, mainly relying on the dependencies between the files. They presented a fully automatic way of mapping vulnerabilities to software components. They analyzed the Mozilla code base to evaluate their approach using SVM for classification. Although their results are very

promising, we could not locate their tool online. Contrary to this work, we predict vulnerabilities at the level of JavaScript functions and we also applied multiple machine learning approaches.

Shin et al. [142] created an empirical model to predict vulnerabilities from source code complexity metrics. Their model was built on the function level similar to ours, but they consider only the complexity metrics. They concluded that vulnerable functions have distinctive characteristics separating them from "non-vulnerable but faulty" functions. They studied the JavaScript Engine from the Mozilla application framework. We use several metrics (including complexity metrics) as predictors and build our prediction models specifically for JavaScript programs. They performed an empirical case study [141] on two large code bases: Mozilla Firefox and Red Hat Enterprise Linux kernel, investigating if software metrics can be used in vulnerability prediction. They considered complexity, code churn, and developer activity metrics. The results showed that the metrics are discriminative and predictive of vulnerabilities. However, their model was also built on file level, while we are predicting vulnerable functions.

Chowdhury et al. [39] created a framework that can predict vulnerabilities mainly relying on the CCC (complexity, coupling, and cohesion) metrics [38]. They compared four statistical and machine learning techniques (namely C4.5 Decision Tree, Random Forests, Logistic Regression, and Naive Bayes classifier). Their case study was also built on the code-base of Mozilla Firefox and used file-level granularity. The authors concluded that decision-tree-based techniques outperformed statistical models in their case. We also found that tree-based classifiers perform well for vulnerable JavaScript function prediction.

Morrison et al. [112] built a model – replicating the vulnerability prediction model by Zimmermann et al [163] – for both binaries and source code at the file level. They figured out that vulnerability prediction at the binary level is not actionable as it would take too much time to inspect the flagged binaries. The authors checked several learning algorithms including SVM, Naive Bayes, Random Forests, and Logistic Regression. On their dataset, Naive Bayes and Random Forests performed the best. In our setup, the Naive Bayes algorithm was the worst performer, while Random Forest achieved good results.

Yu et al. introduced HARMLESS [162], a cost-aware active learner approach to predict vulnerabilities. They used a support vector machine based prediction model with under-sampled training data, and a semi-supervised estimator to estimate the remaining vulnerabilities in a codebase. HARMLESS suggests which source code files are most likely to contain vulnerabilities. They also used Mozilla's codebase in their case study, with 3 different feature sets: metrics, text, and the combination of text mining and crash dump stack traces. The same set of source code metrics were used as that of Shin et al. [141].

In their literature review, Catal et al. [33] concluded that there is a lack of usage of deep neural networks in the field of software vulnerability prediction. We compare many machine learning algorithms for vulnerability prediction, including deep neural networks.

4.2.2 Issue prediction using call graphs

The first papers dealing with call graphs date back to the 1970s [52, 139, 65]. Call-graphs can be divided into two subgroups based on the method used to construct them:

dynamic [158] and static [116].

Dynamic call graphs can be obtained by the actual run of the program. During the run, several runtime information is collected about the interprocedural flow [50]. Techniques, such as instrumenting the source code, can be used for dynamic call graph creation [65, 46].

In contrast, there is no need to run the program in the case of static call graphs, as they are produced by a static analyzer that analyzes the source code of software without actually running it [65]. On the other hand, static call graphs might include false edges (calls) since a static analyzer identifies several possible calls between functions that are not feasible in the actual run of a program; or they might miss real edges. Static call graphs can be constructed from the source code in almost any case, even if the code itself is not runnable.

Different analysis techniques are often combined to obtain a hybrid solution, which guarantees a more precise call graph, thus a more precise analysis [51]. In our work, we also followed this approach as we experienced that using only static call graphs might be imprecise.

With the spread of scripting languages, such as Python and JavaScript, the need for analyzing programs written in these languages also increased [54]. However, constructing precise static call graphs for dynamic scripting languages is a very hard task that is not fully solved yet [161]. The *eval()*, *apply()*, and *bind()* constructions of the languages make it especially hard to analyze the code statically. There are several approaches to construct such static call graphs for JavaScript with varying success [54, 32, 58, 165]. However, the most reliable method is to use dynamic approaches to detect such call edges. We decided to use both dynamic and static analysis to ensure better precision, even though it increases the analysis time, and the code should be in a runnable state due to the dynamic analysis.

Wei and Ryder presented blended taint analysis for JavaScript, which uses a combined static-dynamic analysis [155]. By applying dynamic analysis, they could collect information for even those situations that are hard to analyze statically. Dynamic results (execution traces) are propagated to a static infrastructure, which embeds a call graph builder as well. This call graph builder module makes use of the dynamically identified calls. However, in the case of pure static analysis, they wrapped the WALA tool [4] to construct a static call graph. As previously said, our approach works similarly and also supports additional call graph builder tools to be included in the flow of the analysis.

Feldthaus et al. presented an approximation method to construct a call graph [54] by which a scalable JavaScript IDE support could be guaranteed.

We used a static call graph builder tool in our toolchain, which is based on this approximation method.[5] Additional static JavaScript call graph building algorithms were evaluated by Dijkstra [45]. Madsen et al. focused on the problems induced by libraries used in the project [97]. They used pointer analysis and a novel “use analysis” to enhance scalability and precision.

There are also works intending to create a framework for comparing call graph construction algorithms [91, 17]. However, these are done for algorithms written in Java and C. Call-graphs are often used for preliminary analysis to determine whether an optimization can be done on the code or not. Unfortunately, as they are specific to

[4] `https://github.com/wala/WALA`

[5] `https://github.com/Persper/js-callgraph`

Java and C, we could not use these frameworks for our paper.

Clustering call graphs can have advantages in malware classification [83]. They can help to localize software faults [49], not to mention the usefulness of call graphs in debugging [136].

Musco et al. [117] used four types of call graphs to predict the software elements that are likely to be impacted by a change in the software. However, they used mutation testing to assess the impact of a change in the source code. The same methodology could have been used but with a slight change: instead of using an arbitrary change, it can be a vulnerability introducing or a vulnerability mitigating change.

Nuthan Munaiah and Andrew Meneely [115] introduced two novel attack surface metrics with their approach, which are the "Proximity" and "Risky Walks" metrics. Both of them are defined by the call graph representation of the program. Their empirical study proved that using their metrics to build a prediction model can help to predict more accurately, as their metrics are statistically significantly associated with the vulnerable functions.

Nguyen et al. [121] proposed a model to predict vulnerable components based on a metric set generated from the component dependency graph of a software.

Cheng et al. [37] presented a new approach to detect control-flow-related vulnerabilities called VGDetector. They applied a recent graph convolutional network to embed code fragments in a compact representation (while the representation still preserves the high-level control-flow information).

Lee at al. [89] proposed a new approach to generate semantic signatures from programs to detect malware. They extracted the call graph of the API call sequence that would be generated by the malware, called code graph. This graph is used for the semantic signature. They used semantic signatures to detect malware even if the malware is obfuscated or if it slightly differs from its previous versions (these are the main reasons why a commercial anti-virus does not detect them as malware).

Punia et al. [131] presented a call graph based approach to predict and detect defects in a given program. They also defined call graph based metrics such as *Fan In, Fan Out, Call-Graph Based Ranking (CGBR) and Information Flow Complexity (IFC)*. They investigated the correlation between their metrics and several types of bugs. They proved the hypothesis that there is a correlation between call graph based metrics and bugs in software design. The authors performed their study in the Java domain; contrarily, we focused on JavaScript systems. Besides J84, LMT, and SMO, we applied additional machine learning algorithms and also evaluated deep learning techniques to find potential bugs in the software.

Like many studies, we also focus on different source code metrics, however, we use dynamic analysis (to produce dynamic call graphs) to enhance our model in predicting software issues, we adopt coupling metrics for the so-called hybrid call graph.

4.3 Vulnerability Prediction with Static Source Code Metrics Only

In this section, we present the methodology and the results we obtained in predicting vulnerable JavaScript functions. To build machine learning models, we needed a training dataset with features of JavaScript functions manually labeled as vulnerable or non-vulnerable. The overview of the data mining process we performed is shown in Figure 4.1.

4.3.1 Approach

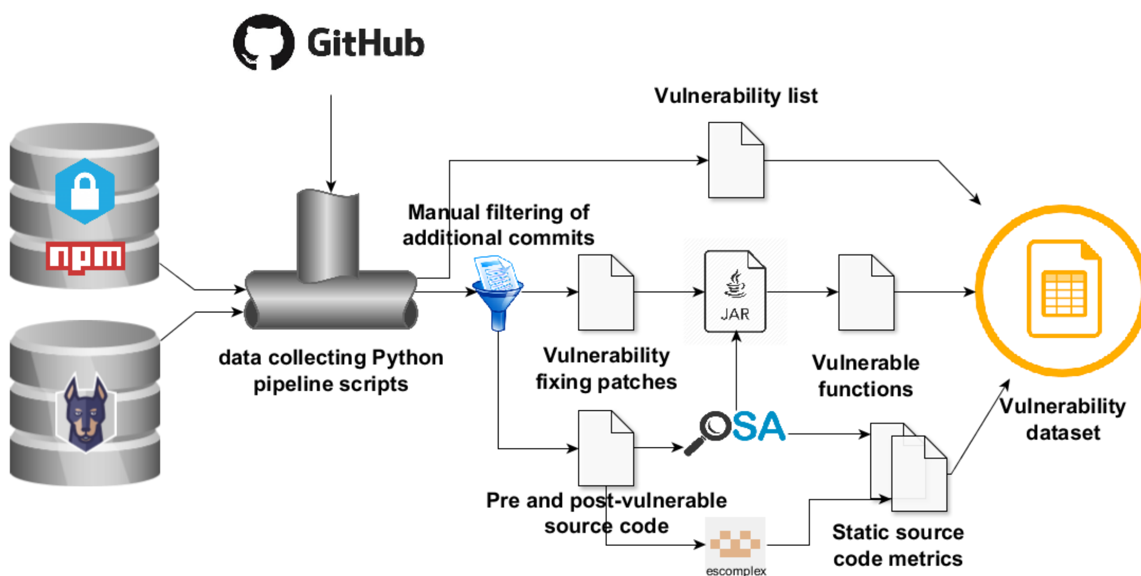


Figure 4.1. Data processing overview

Processing nsp and Snyk and linking them with GitHub We leveraged two publicly available vulnerability databases, nsp (the Node Security Platform, which is now part of npm) [7] and the Snyk Vulnerability Database [12]. Both of these projects aim to analyze programs for vulnerable third party module usages. They have command line and/or web-based interfaces, which can inspect an arbitrary Node.js module to find external dependencies with known vulnerabilities. To achieve this, they utilize a list of known vulnerabilities to look for security issues in the particular version of an external module the programs depend on. We extracted and processed these vulnerability databases.

As for nsp, we used its command line interface to collect vulnerability data. It provides a *gather* command that saves its internal list of vulnerabilities into a JSON file. Snyk has an online repository of known vulnerabilities, but there is no possibility for downloading the entries. Nonetheless, Snyk maintains a GitHub mirror[6] of its vulnerability database with monthly synchronization. We used the content of this GitHub repository in the case of Snyk (accessed on 27/05/2018).

[6] `https://github.com/snyk/vulnerabilitydb`

The main issue with these extracted raw vulnerability sources is that they contain unstructured data. The entries include human-readable descriptions of vulnerabilities with URLs of fixing commits, pull requests or issues in GitHub or other repositories. However, these URLs are somewhat arbitrary; they can appear in multiple places within the entries and any of them might be missing entirely. To handle this, we wrote a set of Python scripts to process these vulnerability entries and create an internal augmented and structured representation of them. The scripts collected all the URLs from each entry $vuln_i$ and kept all those pointing to GitHub. Traversing these URLs, we derived a set of fixing commits (commits pointing to the state of the system where a security vulnerability has been fixed, thus they already contain the mitigation code) for each $vuln_i$ using the GitHub REST API following these steps:

1. If the URL pointed to a particular commit, we put the appropriate commit hash to the fixing list (fix_i)
2. If the URL pointed to a pull request or merge request, we put all the commit hashes in the request to fix_i
3. If the URL pointed to an issue, we traversed through the comments of the issue and collected all mentioned URLs into a separate list for manual validation.

If the separate list for manual validation was not empty, we manually checked all the commit URLs in it and put only those commits into fix_i that were indeed related to the fix of the original vulnerability issue ($vuln_i$). The manual validation was performed by one of the researchers who participated in this study, while another researcher of the team participated in the discussion and resolution of problematic cases. The added commits usually introduced unit tests or some corrections if the first fix was incomplete.

We note that it is possible that a commit that was referenced in the dataset entry (i.e. fix_i) contained tangled code changes (i.e. pull or merge requests). To lower the risk, we performed a random cross-check on several of these large commits, but found no tangled changes in our sample.

Upon finalizing the fixing commit lists for each entry, we collected all their code modifications in the form of a combined patch file ($patch_i$) that contained all the modifications from the fixing commits. We used the GitHub API again to collect this information. Moreover, we identified the parent commit of the first commit in time belonging to the vulnerability fix (sha_{pre}) for each system. Version sha_{pre} was used to assign the labels 1 or 0 to functions indicating whether the function contained a vulnerability or not. The final dataset was assembled from all the sha_{pre} versions of the functions in the systems. We marked all functions that were affected by any of the vulnerability fixing modifications (i.e. $patch_i$ changed those functions) as vulnerable. All the other functions of the JavaScript programs were marked as non-vulnerable. We note that all the test functions (i.e. functions contained in files under "test" folders) have been filtered out as these would only distort the prediction models.

Mapping patches with JavaScript functions To perform the mapping of patches to functions, we used the patch files ($patch_i$ for each vulnerability $vuln_i$ collected by the process described in Section 4.3.1) of the vulnerability-fixing commits in a unified diff format. Each diff contains a header information specifying the name of the original and the new files. After that, there are one or more change hunks that contain the actual line differences and each hunk begins with range information about the modification.

We checked whether any function falls into this range. We achieved this by using the source code positions of the functions – begin and end line numbers, which were produced by the OpenStaticAnalyzer tool – and checked whether these two ranges intersect or not. An example is shown in Listing 4.1.

Listing 4.1. Example diff file

```
--- /path/to/original.js    timestamp
+++ /path/to/new.js    timestamp
@@ -4,1 +4,2 @@
+   var tmp = bar(i);
+   return tmp;
-   return bar(i);
```

Listing 4.2. Example JavaScript function

```
function foo(a) {
  var i = 4 * a;
  // call bar
  var tmp = bar(i);
  return tmp;
}
```

The source position of the *foo* function is [1,6] and the range from the diff is [4,5]. They intersect, so our method incorporates the *foo* function into the dataset. With this algorithm, we found all the functions that were changed by each vulnerability fixing commit, which we mapped to their previous versions (in sha_{pre}) to mark them vulnerable in the version prior to the first fixing commit.

Static source code metrics For predictors (or, features), we used static source code metrics. We calculated the metrics for the functions included in the final dataset using two tools, escomplex [3] and OpenStaticAnalyzer (OSA) [8]. Both OpenStaticAnalyzer [125] and escomplex [35, 101] were used and referenced in related research works, thus we consider them to be reliable. The list of calculated metrics is shown in Table 4.1. Please note that similar metrics are grouped together in one line, so the total number of calculated metrics is 35.

Dataset structure The final dataset structure follows a simple CSV format that is easy to feed into many machine learning frameworks. Each line of the CSV file represents a function from a Node.js program. The 1st column is a short name, while the 2nd is the qualified name of the function generated by the algorithm described in Section 4.3.1. The 3rd column shows the path of the containing JavaScript source file, while the 4th column contains a GitHub URL to the analyzed JavaScript source file (in the sha_{pre} version). The 5th and 6th columns contain the starting, while the 7th and 8th the ending line and column information, respectively. Columns 9 to 43 contain the calculated metric values listed in Table 4.1. The last column (column 44) contains the flag indicating whether the function is vulnerable or not.

[7]Total means that the metric is calculated for the actual code element including all the contained elements recursively.

Table 4.1. Calculated static source code metrics

Metric	Description	Tool
CC	Clone Coverage	OSA
CCL	Clone Classes	OSA
CCO	Clone Complexity	OSA
CI	Clone Instances	OSA
CLC	Clone Line Coverage	OSA
LDC	Lines of Duplicated Code	OSA
McCC, CYCL	Cyclomatic Complexity	OSA, escomplex
NL	Nesting Level	OSA
NLE	Nesting Level without else-if	OSA
CD, TCD	(Total[7]) Comment Density	OSA
CLOC, TCLOC	(Total) Comment Lines of Code	OSA
DLOC	Documentation Lines of Code	OSA
LLOC, TLLOC	(Total) Logical Lines of Code	OSA
LOC, TLOC	(Total) Lines of Code	OSA
NOS, TNOS	(Total) Number of Statements	OSA
NUMPAR, PARAMS	Number of Parameters	OSA, escomplex
HOR_D	Nr. of Distinct Halstead Operators	escomplex
HOR_T	Nr. of Total Halstead Operators	escomplex
HON_D	Nr. of Distinct Halstead Operands	escomplex
HON_T	Nr. of Total Halstead Operands	escomplex
HLEN	Halstead Length	escomplex
HVOC	Halstead Vocabulary Size	escomplex
HDIFF	Halstead Difficulty	escomplex
HVOL	Halstead Volume	escomplex
HEFF	Halstead Effort	escomplex
HBUGS	Halstead Bugs	escomplex
HTIME	Halstead Time	escomplex
CYCL_DENS	Cyclomatic Density	escomplex

The created vulnerability dataset[8] consists of 12,125 JavaScript functions from which 1,496 are vulnerable.

Dataset analysis approach

We employed 8 different types of machine learning algorithms on the vulnerability dataset. These algorithms were

- Logistic Regression Classifier (Logistic) – Logistic regression is a statistical model that uses a logistic function to model a binary dependent variable.
  - Implemented by `sklearn.linear_model.LogisticRegression`
- Naive Bayes Classifier (Bayes) – Naive Bayes classifier is a simple "probabilistic classifier" based on applying Bayes' theorem with strong (naïve) independence assumptions between the features.

[8] `http://www.inf.u-szeged.hu/~ferenc/papers/JSVulnerabilityDataSet/`

  - Implemented by `sklearn.naive_bayes.GaussianNB`

- Decision Tree Classifier (Tree) – Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression, where the goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.
  - Implemented by `sklearn.tree.DecisionTreeClassifier`, an optimized version of the CART algorithm
- Linear Regression Classifier (Linear) – Linear regression is a linear approach to modeling the relationship between a scalar response and one or more explanatory variables also known as dependent and independent variables.
  - Implemented by `sklearn.linear_model.LinearRegression`
- Standard DNN Classifier (DNN_s) – A deep neural network (DNN) is an artificial neural network (ANN) with multiple layers between the input and output layers.
  - Implemented by `tensorflow.layers.dense`
- Customized DNN Classifier (DNN_c) – A custom version of the standard DNN implementing the early stopping mechanism, where we do not train the models for a fixed number of epochs, rather stop when there is no more reduction in the loss function.
  - Implemented by `tensorflow.layers.dense`
- Support Vector Machine Classifier (SVM) – Support-vector machine is a supervised learning model, which is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible.
  - Implemented by `sklearn.svm.SVC`
- K Nearest Neighbors Classifier (KNN) – The k-nearest neighbors algorithm (k-NN) is a non-parametric method for classification and regression, where the input consists of the k closest training examples in the feature space.
  - Implemented by `sklearn.neighbors.KNeighborsClassifier`
- Random Forest Classifier (Forest) – Random forest is an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean/average prediction (regression) of the individual trees.
  - Implemented by `sklearn.ensemble.RandomForestClassifier`

The deep neural network algorithms were implemented in the *TensorFlow* [14] framework[9], while we used *scikit-learn*[10] to run all the other algorithms. Both frameworks were used in a Python environment. We could not use only one of them because while TensorFlow has a strong support for deep learning, it does not contain all the classic algorithms. In contrast, scikit-learn is very strong in classic machine learning algorithms, but it is not a deep learning framework in itself.

[9] `https://www.tensorflow.org/`
[10] `http://scikit-learn.org/stable/`

DNN_s stands for the base DNN algorithm implemented in TensorFlow. We used it without any modifications except for changing the parameters it provides (see Section 4.3.1). DNN_s learning runs for a fixed number of iterations over all the training instances (i.e. epochs). DNN_c is our own modified strategy for training a DNN. It uses an adaptive learning rate method where the learning rate parameter is not constant over the course of training. We start with a relatively high learning rate parameter and continue the classic back propagation algorithm until there is no improvement in the value of F-measure (we call this a *miss*). Then we reduce the learning rate parameter to half, restore the previous model state, and continue the learning process from there. We repeat these steps until we get 4 misses in succession, then terminate the algorithm and return the last, best performing model. This strategy reduces the likelihood of the algorithm getting "stuck" in a local optimum. Regarding KNN, Tree, SVM, Forest, Logistic and Linear regression, and the Naive Bayes algorithm, we used their scikit-learn implementation.

Grid search for the best parameters To find the best performing configuration of each algorithm, we applied a grid search approach [27] on the hyper parameters of the learning algorithms. It means that we defined various values for machine learning algorithm parameters and trained multiple models using various combinations of hyper parameters. After having multiple results for each model, we could select the best performing ones.

For all training sessions, we divided the training data into three sets, *train*, *dev*, and *test* in a 80%, 10%, 10% proportion, respectively, and used 10-fold cross-validation. At the end of the 10 folds, we calculated the precision, recall, and F-measure values. For selecting the best performing parameter configurations, we relied only on the results of the dev set. This ensured that we did not use the information for selecting the best parameters from our final test set in any way. We used F-measure as our primary performance indicator, as in the security domain both precision and recall are important.

Sampling strategies In our assembled vulnerability dataset, only slightly more than 10% of the functions were marked as vulnerable. This highly imbalanced nature of the training set is usually unwanted, as prediction models might be distorted by these skewed distributions.

A common way of handling such situations is the usage of random under or over-sampling strategies [24]. Random under-sampling means we randomly throw away training instances from the larger set until we reach a pre-defined ratio between the two classes. Random over-sampling is when we randomly repeat training instances from the smaller set until we reach a pre-defined ratio between the two classes.

We repeatedly ran our algorithm parameter grid search with the following re-sampling strategies: *no re-sampling* (None); *over-sampling* (↑) with ratios 25%, 50%, 75% and 100%; *under-sampling* (↓) with ratios 25%, 50%, 75% and 100%. For instance, a 50% over-sampling strategy means that we randomly repeat training instances from the vulnerable class until we reach 50% in number compared to the training instances in the non-vulnerable class. Analogously, a 50% under-sampling strategy means we leave out training instances from the non-vulnerable class until it contains only twice as many instances as the vulnerable class. It is important to note here that while over-sampling does not result in loss of training data, under-sampling reduces the number

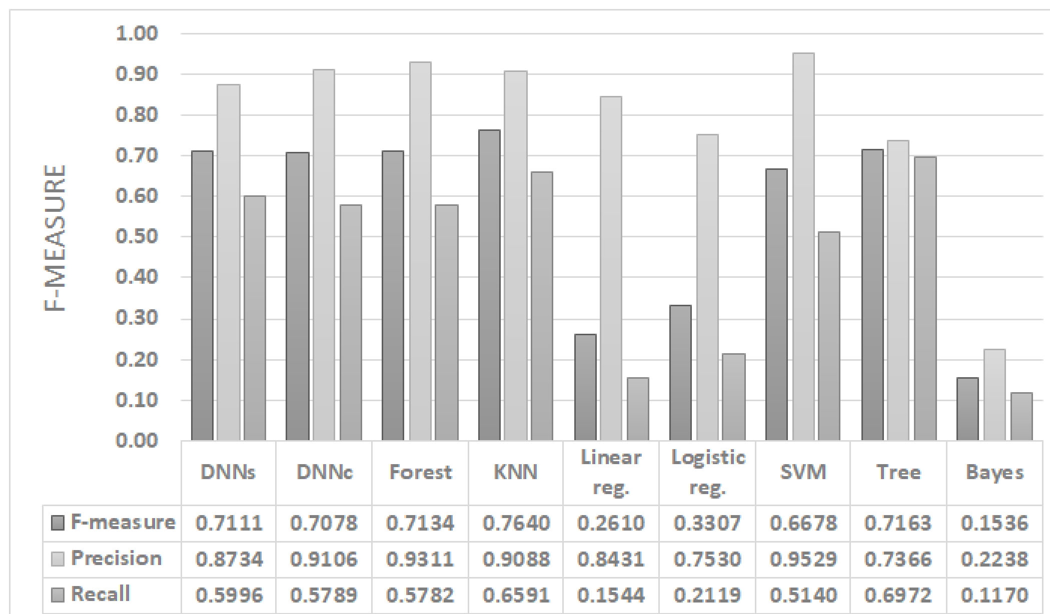
of used training samples.

4.3.2 Results

We trained 9 different prediction models on the created vulnerability dataset (8 different algorithms, but two variants of DNN) on a desktop PC[11] using both CPU and GPU. The running times varied between 6-12 hours for a complete hyper-parameter grid-search of all algorithms. We repeated these grid-search sessions for all the separate over and under-sampling strategies (described in Section 4.3.1), thus building all the models took a considerable amount of time and computing resources.

Results on the imbalanced dataset

First, we ran our grid-search without applying any re-sampling on the vulnerability dataset, which is highly imbalanced (out of 12,125 functions only 1,496 are vulnerable). The performances of the 9 models with their best parameter combinations are displayed in Figure 4.2.



	DNNs	DNNc	Forest	KNN	Linear reg.	Logistic reg.	SVM	Tree	Bayes
F-measure	0.7111	0.7078	0.7134	0.7640	0.2610	0.3307	0.6678	0.7163	0.1536
Precision	0.8734	0.9106	0.9311	0.9088	0.8431	0.7530	0.9529	0.7366	0.2238
Recall	0.5996	0.5789	0.5782	0.6591	0.1544	0.2119	0.5140	0.6972	0.1170

Figure 4.2. Results on the imbalanced dataset

The overall results are surprisingly good given the fact that JavaScript is a highly dynamic language and we only used static source code metrics as predictors. Five out of the 9 models (DNN_s, DNN_c, Forest, KNN, and Tree) achieved an F-measure of over 0.70 and SVM was also very close with 0.67. It is interesting to note that for all algorithms, precision values were significantly higher than recall, except for the decision tree classifier, which had a precision of 0.74, a recall of 0.7 and an F-measure of 0.72.

Only the Naive Bayes algorithm was clearly incapable of producing a viable prediction model using the original, imbalanced vulnerability dataset. Logistic and linear regression achieved a precision of 0.75 and 0.84, respectively, which are relatively high values, however, they had a very low recall (0.21 and 0.15, respectively) that decreased the F-measure values.

[11] 8 core 2.4GHz CPU, NVIDIA Titan Xp GPU, 8GB RAM

As a simple baseline, we also ran the ZeroR algorithm with the setup that made it predict all instances to be vulnerable (and not vice versa as the default setup would do, because if we predicted every instance to be non-vulnerable, all our IR metrics would have been 0). ZeroR achieved a precision of 0.12 and a perfect recall of 1 (as it found all the vulnerable instances), which adds up to an F-measure of 0.21. This result is much worse than those of the other algorithms' except for the Naive Bayes.

Therefore, by choosing a suitable algorithm with proper parameters, it is possible to create efficient function level vulnerability prediction models using only static source code metrics as predictors. The DNN, KNN, Forest, and Tree algorithms all achieved F-measures above 0.70 without any re-sampling on the dataset.

Table 4.2. F-measures achieved by the machine learning algorithms

Alg.	None	↑25%	↑50%	↑75%	↑100%	↓25%	↓50%	↓75%	↓100%	Rand
DNN_s	0.71	**0.71***	0.71	0.65	0.68	0.70	0.71	0.69	0.59	0.05
DNN_c	0.71	0.70	0.71	0.68	0.65	**0.71***	0.71	0.68	**0.66**	0.01
Forest	0.71	**0.74***	**0.74**	**0.73**	**0.72**	0.72	0.72	0.72	0.65	0.05
KNN	**0.76***	**0.75**	0.72	0.6935	0.6817	**0.76**	**0.75**	**0.74**	0.64	0.14
Linear	0.26	0.48	**0.55***	0.49	0.45	0.30	0.37	0.51	0.44	0.02
Logistic	0.33	0.50	**0.57***	0.55	0.49	0.38	0.45	0.53	0.49	0.01
SVM	0.67	0.70	**0.72***	0.70	0.68	0.67	0.67	0.67	0.65	0.16
Tree	**0.72***	0.71	0.71	0.71	0.70	0.70	0.69	0.67	0.59	0.15
Bayes	0.15	0.16	0.16	**0.21***	0.20	0.16	0.16	0.18	0.17	0.07
Median	0.71	0.70	**0.71***	0.68	0.68	0.70	0.69	0.67	0.59	0.05

Comparison of the models based on the complete results

The best performing model results based on the complete grid-search using various re-sampling strategies are summarized in Table 4.2. Each column of the table contains model results (in terms of F-measure[12]) using a particular re-sampling strategy (see Section 4.3.1) with the best parameters found by the grid-search method. The first column shows the results on the original imbalanced dataset without re-sampling (in line with Figure 4.2). The next four columns display the results on the over-sampled, while the following four on the under-sampled dataset. The last column presents results on a random sanity check. To make sure that having these strong prediction results is not coincidental, we created a new training dataset by reassigning the 1,496 vulnerable labels randomly. The training results on this randomly labeled dataset show that models cannot learn to distinguish an arbitrary set of functions based on their static source code metrics, thus our prediction results are unlikely to be the consequences of random factors.

The gray cells in the table mark the best performing algorithm with the given re-sampling strategy. KNN is the best in five different re-sampling configurations, Forest in three, while DNN_c in one. The values indicated in bold and with an asterisk are the best F-measure values for a given machine learning algorithm (i.e. the highest value in the row). The most important thing to note here compared to the results on the imbalanced training set is that even SVM achieved a result above 0.70 with an appropriate over-sampling strategy (↑50%, ↑75%). Seven out of the nine models

[12]Matthews correlation coefficients (MCC) were slightly smaller in general, but they showed the same tendency, see the shared dataset for details.

achieved better performances in some of the re-sampling configurations than on the original, imbalanced dataset. The exact composition of precision and recall values leading to these F-measures are visualized in Figure 4.3.

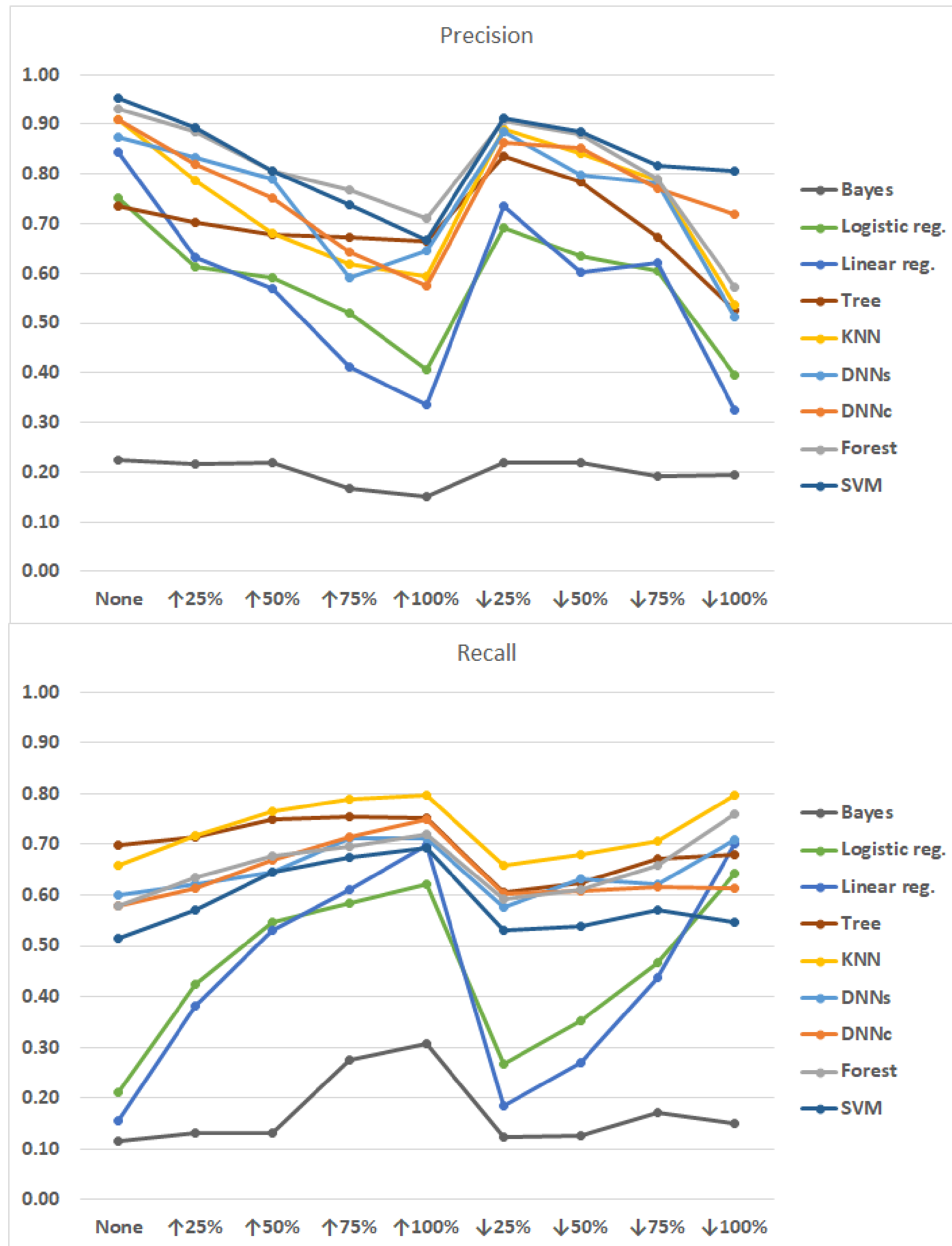


Figure 4.3. Impact of re-sampling on the learning precision and recall

The best performing algorithm for predicting vulnerable JavaScript functions in terms of F-measure was KNN with an F-measure of 0.76 (0.91 precision and 0.66 recall). The best precision (0.95) was achieved by SVM, while the best recall (0.80) by KNN. Overall, KNN, DNN, SVM, Tree, and Forest are equally well-suited for the task, while the regressions, as well as the Naive Bayes algorithm perform much worse.

4.4 Enhancing Bug Prediction with Hybrid Call-Graphs

We got encouraging results using only static code metrics as predictors, despite the highly dynamic nature of JavaScript. The imprecision due to the lack of dynamic information might affect the defect prediction models used on them. We assume that combining static and dynamic analysis improves the performance of the defect prediction models. In this section, we propose a function level JavaScript issue prediction model based on static source code metrics with the addition of hybrid (static and dynamic) code analysis based metrics of the number of incoming and outgoing function calls (HNII and HNOI).

We also created a hybrid call graph framework, namely the *hcg-js-framework*[13]. The framework supports the hybrid code analysis by extracting call graph information of JavaScript functions using both static and dynamic analysis.

Our previously proposed database (introduced in Section 4.3) contains the vulnerabilities and their fixes from 93 different programs. In order to gather dynamic traces of the execution from each software, we would have needed a much more complex workflow. Furthermore, we would have needed to ensure the following conditions:

- All 93 different software can be tested with the `"npm test"` command.
- Install and manage all of the external dependencies (besides the ones mentioned in `package.json`).
- Each and every program has sufficient test coverage.

Providing these conditions requires a huge effort. We also would like to apply our machine learning approach to a slightly different, wider field, namely to predict generic software bugs.

For that, we used a publicly available bug dataset, BugsJS [68]. It consists of 453 real, manually validated bugs and their fixing patches from 10 popular JavaScript projects. From this dataset, we selected the ESLint[14] project to test our hybrid model approach, which contains 333 bugs.

4.4.1 Approach

Our approach consists of numerous steps, which we present in detail in this section. Figure 4.4 shows the steps required to produce input for the machine learning algorithms.

BugsJS Dataset

BugsJS [68] is a bug dataset inspired by Defects4J [80], however, it provides bug related information for popular JavaScript-based projects instead of Java projects. Currently, BugsJS includes bug information for ten projects that are actively maintained Node.js server-side programs hosted on GitHub. Most importantly, BugsJS includes projects which adopt the Mocha testing framework; consequently, we can implement dynamic analysis experiments easier.

BugsJS stores the forks of the original repositories and extends them by adding tags to their custom commits in the form of:

[13] `https://github.com/sed-szeged/hcg-js-framework`

[14] `https://eslint.org/`

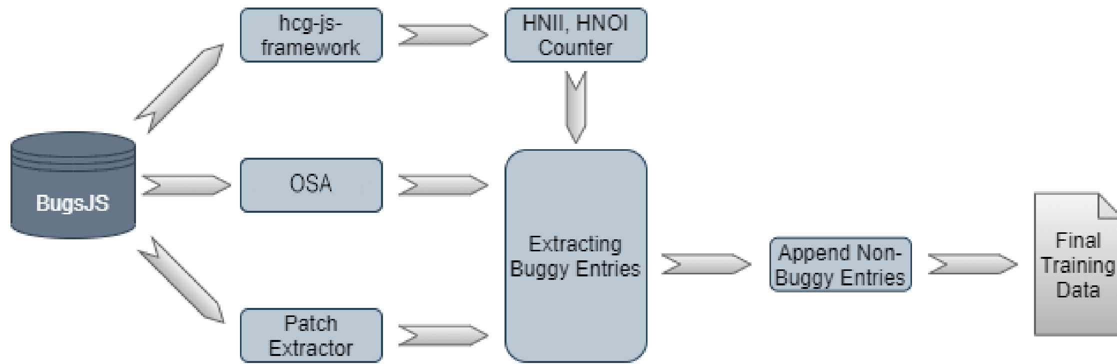


Figure 4.4. The schematic view of our approach

- `Bug-X`: The parent commit of the revision in which the bug was fixed (i.e., the buggy revision)
- `Bug-X-fix`: A revision (commit) containing only the production code changes (test code and documentation changes were excluded) introduced in order to fix the bug

, where `X` denotes a number associated with a given bug. As out of the total 453 bugs, ESLint [15] itself contains 333 bugs, we chose this project as input in our study.

Hybrid Invocation Metrics Calculation

As a first step, we have to produce the so-called *hybrid call graphs* from which we can calculate the *hybrid invocation metrics* (i.e., HNII and HNOI). In order to understand what a hybrid call graph is, let us consider Figure 4.5, which shows the details of the node "hcg-js-framework" presented earlier in Figure 4.4.

As can be seen, the input of the *hcg-js-framework* is the JavaScript source code that we want to analyze, which can be either a Git repository or a local folder. Then we analyze the source code with various static and dynamic tools. Following the analyses, the framework converts all the tool-specific outputs to a unified JSON format. Once we have the JSON files, the framework combines them into a merged JSON. This merged JSON contains every node and edge (JavaScript function nodes and call edges between them) that either of the tools found.

After this step, we augment this merged JSON with confidence levels for the edges. The confidence levels are calculated based on a manual evaluation of 600 out of 82,791 call edges found in 12 real-world Node.js modules. We calculated the True Positive Rate for each tool intersection. We estimate the confidence of a call edge with these rates. For instance, if a call edge was found by tools A and B, and in the manually evaluated sample, there were ten edges found by only these tools, from which five turned out to be a valid call edge, we add confidence of 0.5 to all these edges.

Figure 4.6 shows a Venn diagram of the call edges found in 12 Node.js modules. We have an evaluation ratio for each intersection, which the framework uses for edge confidence level estimation.

To sum it up, a *hybrid call graph* is a call graph (produced by combining the results of both static and dynamic analysis) which associates a confidence factor to each call

[15] `https://github.com/eslint/eslint`

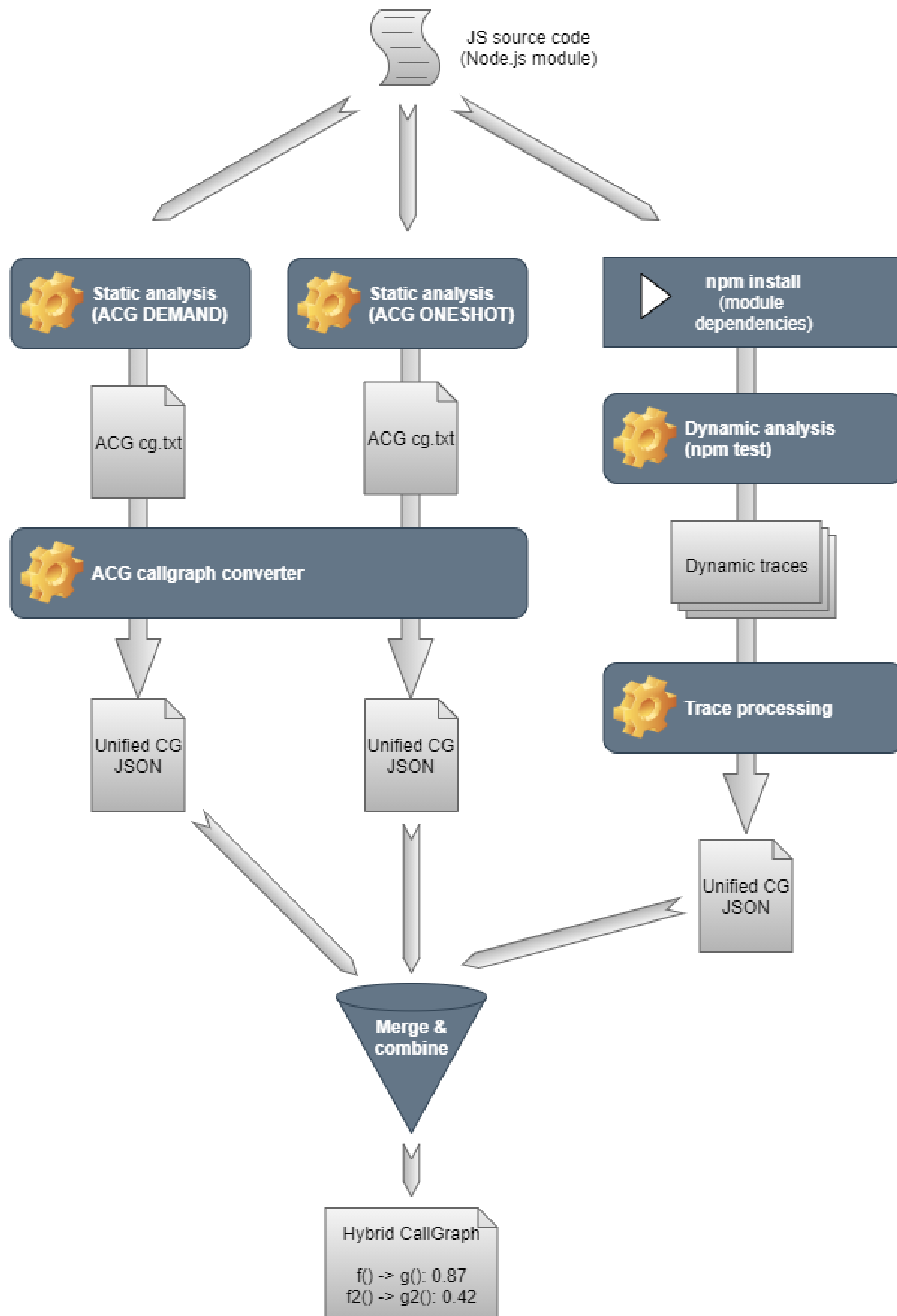


Figure 4.5. Hybrid call graph framework architecture

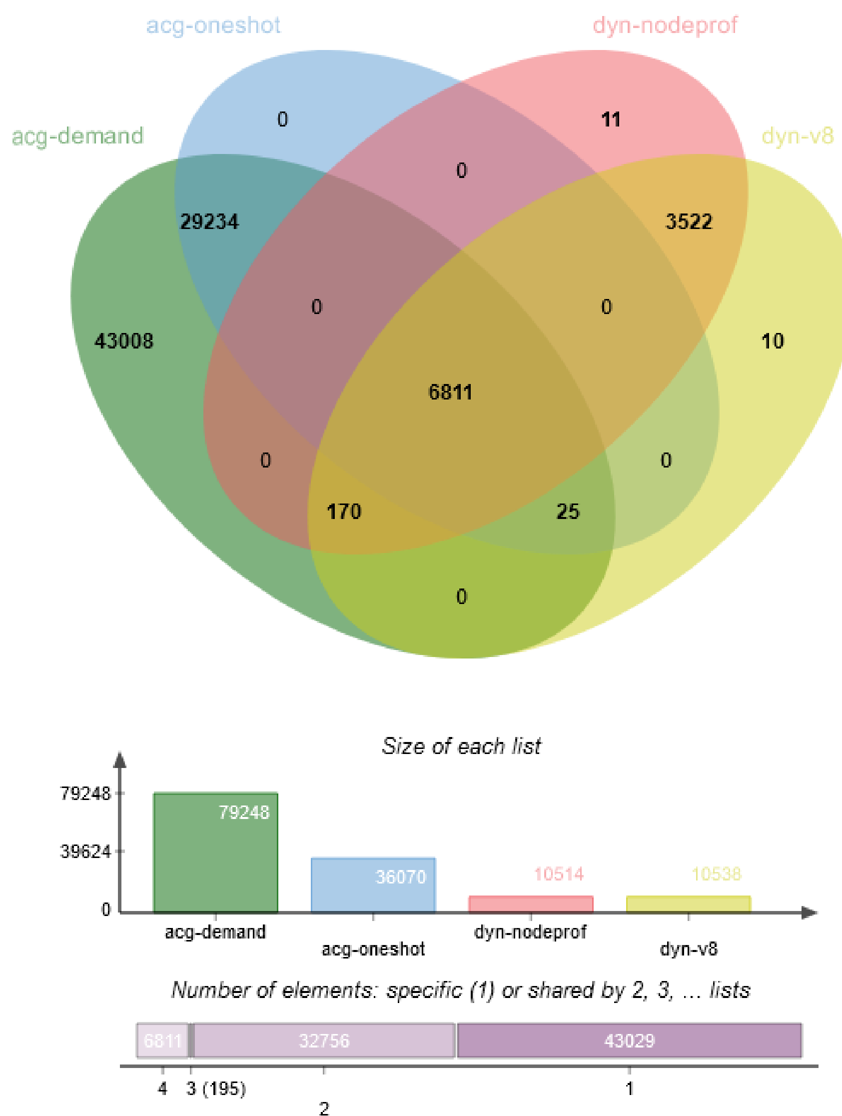


Figure 4.6. Venn diagram of found edges

edge, which shows how likely an edge is to be valid (higher confidence means higher validity).

This hybrid call graph is the input of the *HNII, HNOI Counter* which is responsible for calculating the exact number of incoming and outgoing invocations (i.e., NII, NOI). At this point, we have to specify the *threshold* value, which defines the lower limit from which we consider a call edge to be a valid call edge, thus contributing to the value of the number of incoming and outgoing invocations. We considered four threshold values: *0.00*, *0.05*, *0.20* and *0.30*. In the case of the first one, all edges are considered as possibly valid call edges, while the latter ones only include edges with a higher confidence factor. We name these two new metrics as HNII (Hybrid Number of Incoming Invocations) and HNOI (Hybrid Number of Outgoing Invocations) to differentiate them from the original static NII and NOI metrics.

The *HNII, HNOI Counter* traverses all call edges and considers those edges where the confidence level is above the given threshold. The edges fulfilling this threshold criteria contribute to the HNOI metric of the source node and the HNII metric of the target node. As a result, the tool produces a JSON file as its output, which contains only the nodes (i.e., the JavaScript functions) with their corresponding HNII, HNOI metric values, and additional information about their position in the system, such as source file, line, and column. Listing 4.3 shows an example of a single node output.

Listing 4.3. Sample output from the HNII, HNOI Counter

```
1 {
2     "pos": "eslint/lib/ast-utils.js:169:25",
3     "entry": false,
4     "final": false,
5     "hnii": 1,
6     "hnoi": 3
7 }
```

OpenStaticAnalyzer and Patch Extraction

Besides computing the hybrid metrics, HNII and HNOI, a standard set of metrics is provided by a static source code analyzer named OpenStaticAnalyzer (OSA) [8]. OpenStaticAnalyzer also takes JavaScript source code as input, and outputs (amongst others) different CSV files for different source code elements (functions, methods, classes, files, system).

In this study, we used the resulting CSV file that contains function level entries, which includes static size metrics (LOC, LLOC, NOS), complexity metrics (McCC, NL), documentation metrics (CD, CLOC, DLOC), and traditional coupling metrics (NII, NOI) as well. The exact definition of the metrics can be found in Table 4.1. These metrics are calculated for all the 333 bugs in ESLint before and after the bug is fixed, which means 666 static analyses in total. Similarly, we extracted the patches for these 333 bug fixing commits, which is done by *Patch Extractor*.

Composing Buggy Entries

At this point, we have all the necessary inputs to combine them in one CSV, which contains the buggy entries with their static source code metrics extended with the

HNII and HNOI metrics. The core of the algorithm is the following. We traverse all the bugs one-by-one. For bug_i, we retrieved a set of entries from the ith static analysis results, which were touched by the fixing $patch_i$ (determined based on entry name and positional information), and extended these entries with the corresponding HNII and HNOI metric values. We included the before-fix state (i.e., the buggy) for the touched JavaScript functions, and used the date of the latest bug to select non-buggy instances from that corresponding version of the code (i.e., Bug-79 fixed at 2018-03-21 17:23:34). For non-buggy entries, we also extracted the corresponding HNII and HNOI values also from the latest buggy version.

4.4.2 Results

To calculate the HNOI and HNII metrics, one needs to apply a threshold to the call edges (to decide which edges to consider as valid) in the underlying hybrid (also called fuzzy) call graph produced by the *hcg-js-framework* (see Section 4.4.1). We calculated the metric values[16] with four different thresholds: 0, 0.05, 0.20, and 0.30. Table 4.3 shows the descriptive statistics of the metrics on our ESLint dataset.

Table 4.3. Descriptive statistics of the HNII and HNOI metrics calculated using different thresholds

		HNII			**HNOI**	
Threshold	**Avg.**	**Median**	**Std.dev.**	**Avg.**	**Median**	**Std.dev.**
0.00	7.026021	1	26.95583	5.341887	2	27.27586
0.05	6.961330	1	26.95997	5.243224	2	26.91228
0.20	0.840622	1	2.823739	1.018793	0	9.236607
0.30	0.840622	1	2.823739	1.018793	0	9.236607

As can be seen, thresholds 0.20 and above significantly reduces the number of considered edges for HNII and HNOI calculation. We wanted to use as many of the extracted call edges as possible, so we opted to use the 0.00 threshold later on (i.e., we considered each edge in the fuzzy call graph where the weight/confidence is greater or equal to zero).

We trained several models (the models are described in Section 4.3.1) on the dataset with three different configurations for the features:

- Purely static metrics ($S-0_00_s.csv$): the dataset contains only the pure static source code metrics (i.e., original versions of NOI and NII plus all the provided metrics by OpenStaticAnalyzer, see Section 4.4.1).

- Static metrics with only hybrid NOI and NII versions ($H-0_00_h.csv$): the dataset contains all the static metrics except NOI and NII, which are replaced by their hybrid counterparts (HNOII and HNII) calculated on the output of *hcg-js-framework*.

- Both static and hybrid metrics ($S+H-0_00_s+h.csv$): the dataset contains all the static metrics plus the hybrid counterparts of NOI and NII (HNOII and HNII) calculated on the output of *hcg-js-framework*.

[16] All the data used in this study is available online [20]

With the various hyper-parameters, the trained models added up to a total of 36 configurations. We executed all these 36 training tasks on all three feature sets, so we created 108 different machine learning models for comparison. To cope with the highly imbalanced nature of the dataset (i.e., there are significantly more non-buggy functions than buggy ones), we applied a 50% oversampling on the minority class. We also standardized all the metric values to bring them to the same scale. For the model training and evaluation, we used the open-source DeepWater Framework[17] [56], which contains the implementation of all the used algorithms.

To ensure that the results are robust against the chosen threshold, we trained the same 108 models with the threshold value of 0.30 for HNII and HNOI calculation. We found that the differences among S, H, and $S + H$ feature sets became smaller, but the general tendency that H and especially $S + H$ features achieved better results remained. Therefore, in the rest of the chapter, we can use the HNII and HNOI metrics calculated with the 0.00 threshold without the loss of generality. In the rest of this chapter, we present our findings. We would like to note that the abbreviations in the tables are the same as we defined earlier in this chapter, in Section 4.3.1).

The Best Performing Algorithms

Table 4.4. Top 10 recall measures

Alg.	Feat. Set	Accuracy	Precision	Recall ↓	F-measure	MCC
DNN_c	S+H	0.763 (±0.031)	0.595 (±0.056)	0.642 (±0.074)	0.618 (±0.043)	0.447 (±0.063)
KNN	S+H	0.788 (+0.025)	0.646 (+0.051)	0.635 (-0.007)	0.641 (+0.023)	0.490 (+0.043)
KNN	H	0.776 (+0.012)	0.621 (+0.026)	0.631 (-0.011)	0.626 (+0.008)	0.466 (+0.019)
DNN_s	H	0.749 (−0.014)	0.571 (−0.024)	0.631 (−0.011)	0.600 (−0.018)	0.419 (−0.028)
KNN	S+H	0.772 (+0.009)	0.617 (+0.022)	0.619 (−0.023)	0.618 (+0.000)	0.455 (+0.008)
KNN	S	0.763 (+0.000)	0.599 (+0.004)	0.619 (−0.023)	0.609 (−0.009)	0.439 (−0.008)
KNN	S+H	0.787 (+0.024)	0.650 (+0.055)	0.619 (−0.023)	0.634 (+0.016)	0.484 (+0.037)
KNN	S	0.769 (+0.006)	0.613 (+0.018)	0.614 (−0.028)	0.613 (−0.004)	0.449 (+0.002)
KNN	H	0.778 (+0.014)	0.630 (+0.035)	0.613 (−0.029)	0.622 (+0.004)	0.464 (+0.017)
KNN	H	0.768 (+0.004)	0.610 (+0.015)	0.609 (−0.033)	0.610 (−0.008)	0.444 (−0.003)

Table 4.4 contains data about the top 10 models' results based on their recall values. We ranked all 108 models, meaning that all three feature sets are on the same list.

We can measure recall with the following formula:

$$Recall = \frac{TP}{TP + FN}$$

, where TP means True Positive samples, while FN means False Negatives. As we can see, DNN_c (0.642) and KNN (0.635) models achieve the best recall values on the $S + H$ feature set. The same models produce almost as high recall values (0.631) using only the H feature set. The best performing model on the S feature set is KNN, with a significantly lower (0.619) recall value. It shows that hybrid invocation metrics do increase the performance of machine learning models in terms of recall. The best values are achieved by keeping both the original NOI and NII metrics and adding their hybrid counterparts HNOI and HNII, but using only the latter ones as substitutes for the static metrics still improves recall values.

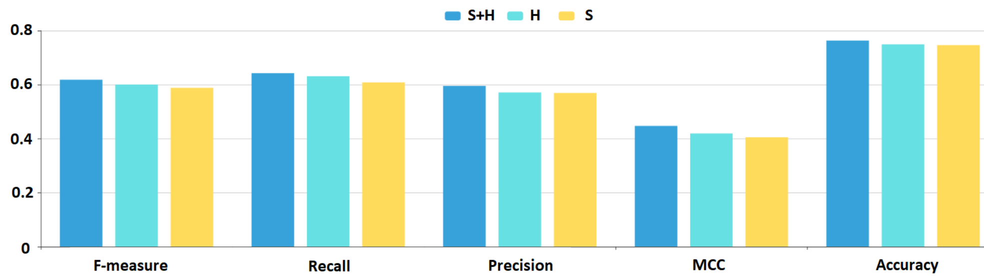


Figure 4.7. The best Deep Neural Network (DNN) configurations for all three feature sets

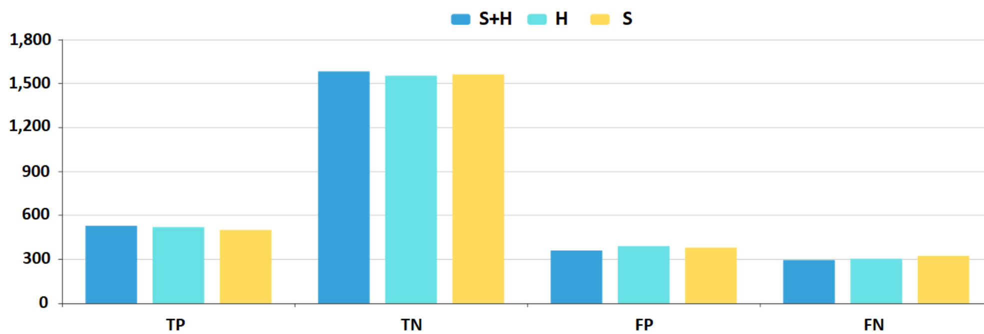


Figure 4.8. The best Deep Neural Network (DNN) configurations for all three feature sets

To visualize the difference in the various performance measures, we plotted a bar-chart (Figures 4.7 and 4.8) with the best DNN configurations (i.e., applying the set of hyper-parameters with which the model achieves the best performance) for all three feature sets. Blue marks the results using the $S + H$ feature set, cyan the H feature set, while yellow the S feature set. $S + H$ results are superior, while H results are better than the S results except for the False Positive and True Negative instances. The chart shows that there is a constant 3-4% improvement in all aspects of the DNN model results if we add the hybrid metrics to the feature sets.

Table 4.5. Top 10 precision measures

Alg.	Feat. Set	Accuracy	Precision ↓	Recall	F-measure	MCC
SVM	H	0.756 (±0.019)	0.829 (±0.069)	0.229 (±0.055)	0.359 (±0.073)	0.348 (±0.069)
SVM	S	0.757 (+0.000)	0.827 (−0.002)	0.232 (+0.002)	0.362 (+0.003)	0.349 (+0.001)
SVM	S+H	0.759 (+0.003)	0.824 (−0.005)	0.244 (+0.015)	0.376 (+0.017)	0.358 (+0.010)
SVM	S	0.759 (+0.003)	0.767 (−0.062)	0.275 (+0.046)	0.405 (+0.046)	0.355 (+0.007)
SVM	H	0.755 (−0.001)	0.762 (−0.067)	0.260 (+0.030)	0.387 (+0.028)	0.341 (−0.007)
SVM	S+H	0.760 (+0.004)	0.756 (−0.073)	0.289 (+0.059)	0.418 (+0.059)	0.359 (+0.011)
Forest	S+H	0.816 (+0.060)	0.753 (−0.076)	0.569 (+0.340)	0.648 (+0.289)	0.536 (+0.188)
Forest	H	0.814 (+0.057)	0.743 (−0.086)	0.573 (+0.343)	0.647 (+0.288)	0.532 (+0.184)
Forest	S+H	0.808 (+0.052)	0.731 (−0.098)	0.564 (+0.335)	0.637 (+0.278)	0.518 (+0.170)
Forest	H	0.810 (+0.053)	0.726 (−0.103)	0.579 (+0.350)	0.644 (+0.285)	0.523 (+0.174)

[17] https://github.com/sed-inf-u-szeged/DeepWaterFramework

Table 4.5 presents the top 10 model results based on their precision values. We ranked all 108 models, meaning that all three feature sets are on the same list. We can measure precision with the following formula:

$$Precision = \frac{TP}{TP + FP}$$

, where TP means True Positive samples, while FP means False Positives. As we can see, the SVM model (0.829) achieves the best precision values on the H feature set. Interestingly, SVM produces an almost as high precision value (0.827) using only the S feature set as well. Based on the $S + H$ feature set, SVM achieves a precision value of 0.824. It shows that hybrid invocation metrics do increase the performance of ML models in terms of precision, but not as significantly as in the case of recall values. Nonetheless, for algorithms other than SVM, the increase is more significant.

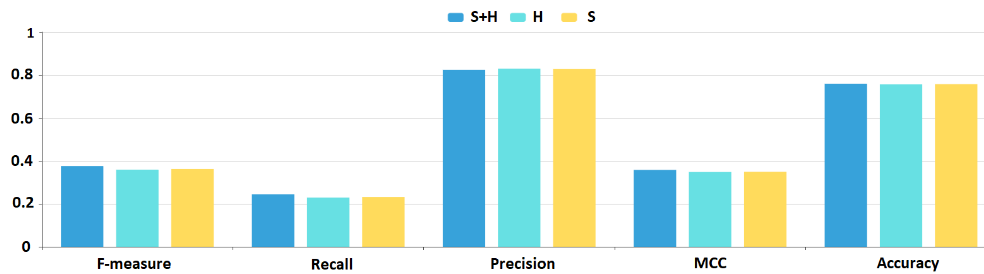


Figure 4.9. The best Support Vector Machine (SVM) configurations for all three feature sets

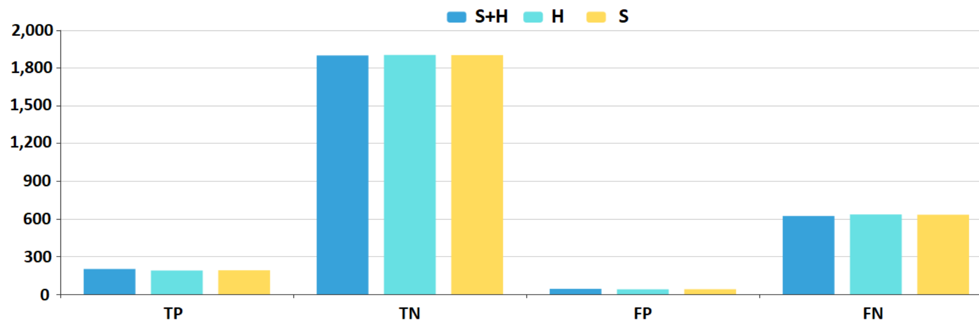


Figure 4.10. The best Support Vector Machine (SVM) configurations for all three feature sets

To visualize the difference in the various performance measures, we plotted a bar-chart (Figures 4.9 and 4.10) with the best SVM configurations for all three feature sets. Blue marks the results using the $S + H$ feature set, cyan the H feature set, while yellow the S feature set. $S+H$ results are superior, while H results are still better than S results for all measures. The chart shows that there is a constant 1-2% improvement in all aspects of the SVM model results if we add the hybrid metrics to the feature sets.

Table 4.6 shows the top 10 model results based on their F-measure values. We ranked all 108 models, meaning that all three feature sets are on the same list. We can

Table 4.6. Top 10 F-measures

Alg.	Feat. Set	Accuracy	Precision	Recall	F-measure ↓	MCC
Forest	S+H	0.816 (±0.024)	0.753 (±0.046)	0.569 (±0.068)	0.648 (±0.054)	0.536 (±0.064)
Forest	H	0.814 (−0.002)	0.743 (−0.010)	0.573 (+0.004)	0.647 (−0.001)	0.532 (−0.005)
Forest	H	0.810 (−0.007)	0.726 (−0.027)	0.579 (+0.010)	0.644 (−0.004)	0.523 (−0.014)
KNN	S+H	0.788 (−0.028)	0.646 (−0.106)	0.635 (+0.066)	0.641 (−0.008)	0.490 (−0.046)
Forest	S+H	0.805 (−0.011)	0.712 (−0.041)	0.579 (+0.010)	0.639 (−0.010)	0.512 (−0.024)
Forest	H	0.799 (−0.017)	0.690 (−0.063)	0.592 (+0.023)	0.637 (−0.011)	0.503 (−0.034)
Forest	S+H	0.808 (−0.008)	0.731 (−0.022)	0.564 (−0.005)	0.637 (−0.011)	0.518 (−0.019)
Forest	S+H	0.799 (−0.017)	0.690 (−0.062)	0.587 (+0.018)	0.635 (−0.013)	0.500 (−0.036)
KNN	S+H	0.787 (−0.029)	0.650 (−0.103)	0.619 (+0.050)	0.634 (−0.014)	0.484 (−0.052)
KNN	H	0.776 (−0.040)	0.621 (−0.132)	0.631 (+0.062)	0.626 (−0.022)	0.466 (−0.071)

calculate F-measure with the following formula:

$$F - measure = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}.$$

As we can see, Random Forest (0.648) and KNN (0.641) models achieve the best F-measures on the $S+H$ feature set. The same Random Forest models produce almost as high F-measures (0.647) using only the H feature set. The best performing model on the S feature set is not even in the top 10. It shows that hybrid invocation metrics do increase the performance of ML models in terms of F-measure, meaning they improve the models' overall performance. The best values are achieved by keeping both the original NOI and NII metrics and adding their hybrid counterparts HNOI and HNII, but using only the latter ones as substitutes for the static metrics still improves F-measure significantly.

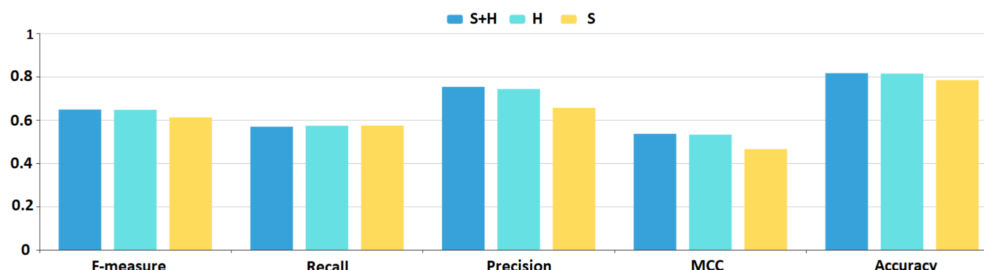


Figure 4.11. The best Random Forest (Forest) configurations for all three feature sets

To visualize the difference in the various performance measures, we plotted a bar-chart (Figures 4.11 and 4.12) with the best Random Forest configurations for all three feature sets. Blue marks the results using the $S + H$ feature set, cyan the H feature set, while yellow the S feature set. $S+H$ and H results are better than S results for all measures except for recall, but the difference there is only marginal. The chart shows that there is a constant 1-2% improvement in all aspects of the Random Forest model results, but precision is higher by approximately 10% if we add the hybrid metrics to the feature sets.

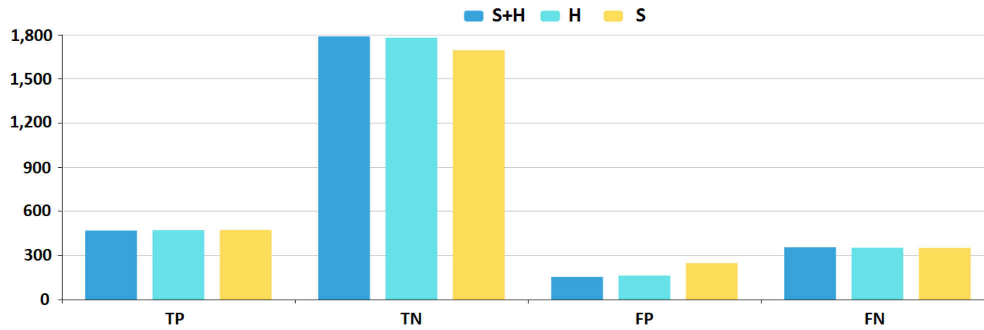


Figure 4.12. The best Random Forest (Forest) configurations for all three feature sets

The Most Balanced Algorithm

K-nearest neighbors models stand out in that they produce the most balanced performance measures. As can be seen in Figures 4.13 and 4.14, both precision and recall values are above 0.6, therefore, F-measure is above 0.6 as well. For this model, the H feature set brings a 1-2% improvement over the S feature set, while the $S + H$ feature set results in a 2-5% increase in performance.

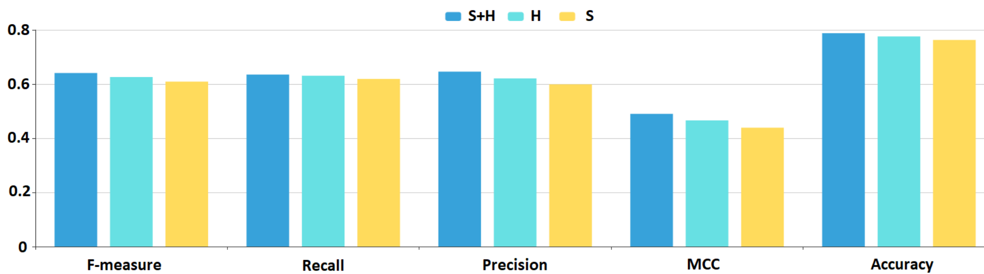


Figure 4.13. The best K-Nearest Neighbors (KNN) configurations for all three feature sets

Significance Analysis of the Performance Measures

Despite a seemingly consistent increase in every model performance measure caused by adding hybrid source code metrics to the features, we could not be sure that this improvement is statistically significant. Therefore, we performed a Wilcoxon signed-rank test [157] on the model F-measure values between each pair of feature sets (S vs. H, S vs. $S + H$, H vs. $S + H$). The detailed results (T statistics and p-values) are shown in Table 4.7.

As can be seen, the F-measure values achieved by the models differ significantly (p-value is less than 0.05) depending on the feature sets we used for training them. It is interesting to observe that there is a significant difference in performances even between the models using the hybrid and static+hybrid features and not just between models using static features only and models using hybrid features as well. These results confirm that even though hybrid source code metrics provide additional prediction power to bug prediction models, they do not substitute static source code metrics

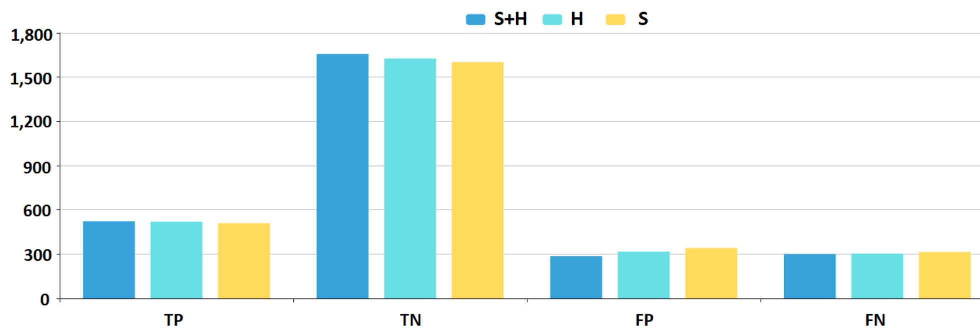


Figure 4.14. The best K-Nearest Neighbors (KNN) configurations for all three feature sets

Table 4.7. Wilcoxon signed-rank test results of F-measures between models using different feature sets

	Hybrid		Static+Hybrid	
Features	**T**	**p-value**	**T**	**p-value**
Static	95.5	0.00032	74	0.00004
Hybrid	-	-	116	0.0019

but complement them. Therefore, according to our empirical data, we could achieve the best performing JavaScript bug prediction models by keeping static call-related metrics and adding their hybrid counterparts to the model features. We note that the Wilcoxon signed-rank test showed significant results in the case of precision and recall performance measures as well.

Discussion

In the previous sections, we analyzed the best performing algorithms with a focus on the improvements caused by the hybrid source code metrics. However, to have a complete picture of the results, we summarize the performances of all nine machine learning algorithms here. Table 4.8 shows the best prediction performances (i.e., models with best performing hyper-parameters and feature sets) of all nine algorithms according to their F-measures.

Table 4.8. The best results of the nine ML models according to their F-measure

ML algorithm	Feature set	Accuracy	Precision	Recall	F-measure	MCC
Random Forest	S+H	0.816	0.753	0.569	0.648	0.54
K Nearest Neighbors	S+H	0.788	0.646	0.635	0.641	0.49
Customized DNN	S+H	0.784	0.649	0.601	0.624	0.47
Decision Tree	S+H	0.781	0.649	0.58	0.612	0.46
Standard DNN	H	0.774	0.634	0.569	0.6	0.44
Logistic Regression	S+H	0.787	0.682	0.533	0.598	0.46
Support Vector Machine	S+H	0.789	0.699	0.515	0.593	0.47
Linear Regression	S+H	0.769	0.67	0.443	0.533	0.4
Naive Bayes	S+H	0.772	0.713	0.394	0.508	0.4

There are several properties to observe in the table. First, all but one machine learning model achieves the best results in terms of F-measure using the $S+H$ feature set. The only exception is the Standard DNN Classifier, which performs best using only the hybrid version of call metrics (i.e., H feature set). Second, the Random Forest Classifier not only has the best F-measure (0.648), but also the best accuracy (0.816), precision (0.753), and MCC (0.54) metrics, which mean it performs the best overall for predicting software bugs in the studied JavaScript program. However, there is a trade-off between precision and recall; therefore, the Random Forest Classifier only has the third highest recall metric (0.569). Third, the K Nearest Neighbors Classifier has the second to last lowest precision (0.646) and only the third-highest accuracy (0.788); still, its recall (0.635) is the highest among all the models and in terms of F-measure (0.641) and MCC (0.49) it is a very close second behind the Random Forest Classifier. This implies that the K Nearest Neighbors Classifier achieves the most balanced performance, not the best in every aspect but very high performance measures with no large variance. Fourth, the deep neural networks (Standard DNN Classifier and Customized DNN Classifier) do not outperform the simpler, classical models in this prediction task. The most likely cause of this is the relatively small amount of training samples, so the real strength of deep learning cannot be exploited. Fifth, the models struggle to achieve high recall values in general, which seems to be the bottleneck of the maximum F-measures. Even the worst-performing models (Linear Regression Classifier and Naive Bayes Classifier) have an acceptable accuracy (0.769 and 0.772, respectively) and precision (0.67 and 0.713, respectively), but very low recall (0.443 and 0.394, respectively), which results in a very low F-measure (0.533 and 0.508, respectively).

To sum up our experiences, it is worthwhile to add hybrid call metrics to the set of standard static source code metrics for training a JavaScript bug prediction model. To achieve the highest accuracy and precision, one should choose the Random Forest Classifier method, but if the recall is also important and one wants to have as balanced results as possible, the K Nearest Neighbors Classifier is the best possible option.

4.5 Threats to Validity

In their work, Basili et al. [23] stated that generalizing results from empirical studies in software engineering is difficult. This is because any process, any study depends on quite a lot of context variables. Our results might be affected by this phenomenon as well and might change with context (e.g. building defect prediction models for other languages).

Our data collection process (described in Section 4.3.1) might not be 100% accurate, as only the additional candidate commits collected from issue comments were validated manually. The original data sources might contain errors as well, and our automatic patch collection and patch-to-function mapping algorithms might introduce inconsistencies. We tried to mitigate this problem by thorough code review of our scripts and programs. In the case of BugsJS, the dataset and the corresponding paper are already published [68]. Therefore, we consider them to be correct.

We mapped static source code analysis results of various tools and functions identified in patches by line information. This is another source of possible errors, but we performed a small evaluation on 20 randomly selected JavaScript functions from the dataset and found no multiple functions in the same line. Based on this and our past

experience, we believe it is a safe assumption that multiple functions in the same code line are very rare in a non-minified JavaScript program. As we used line information only within the same version of the programs, the likelihood of mismatching functions is even more negligible.

The extraction of features (i.e. static source code metrics) is heavily dependent on the accuracy of the tools used, which may threaten the extraction process. However, there are numerous related works using the same tools, thus they can be considered stable. Moreover, we manually double-checked some of the calculated metric values and found no problems in their calculation.

As a training set in Section 4.4 we used 333 bugs from one system. Therefore, the results might be specific to this system and might not generalize well. However, ESLint is a large and diverse program containing a representative set of issues. Additionally, bugs are manually filtered, thus do not introduce noise in the prediction models. As a result, we believe that our study is meaningful, though replication with more subject systems would be beneficial.

The threshold value chosen for calculating the hybrid call edges might affect the ML model performances. We selected a threshold of 0 (i.e., counted every edge with a weight greater than zero) in our case study, however, we carried out a sensitivity analysis with different thresholds as well. Even though the calculated HNOI and HNII values changed based on the applied threshold, the model improvements using these values proved to be consistent with the ones presented in the study. Therefore, we believe that the essence of the results is independent of the choice of the particular threshold value.

Finally, the provided thresholds might be inaccurate as we derived them from a manual evaluation of a small sample of real call edge candidates. To minimize the risk of human error, two senior researchers evaluated all the edges who had to agree on each call label. For sampling, we applied a stratified selection strategy, so we evaluated more call samples from subsets of tools finding more edges in general, thus increasing the confidence of the derived weights.

4.6 Summary

In this chapter, we proposed two, function level JavaScript issue prediction models by challenging 8 well-known, widely used machine learning algorithms, and enhanced the issue prediction with the addition of hybrid (static and dynamic) code analysis based metrics for incoming and outgoing function calls. JavaScript is a highly dynamic scripting language for which static code analysis might be very imprecise, therefore, combining static and dynamic analysis to extract features is a promising approach.

In order to achieve this goal, we created and published a novel JavaScript vulnerability dataset to be used for building prediction models. The dataset contains various JavaScript functions together with their static source code metrics and a flag indicating whether the function contains a vulnerability or not. This information was assembled by mining the public vulnerability data sources, nsp and Snyk, and collecting fixing patches from GitHub. We presented an assessment of existing machine learning algorithms for building function level vulnerability prediction models using this dataset. We analyzed the performances of 8 different types of algorithms using the training set as-is, and also by applying various re-sampling strategies.

Our preliminary results show that even for such a highly dynamic language as JavaScript, static source code metrics are suitable predictors of vulnerabilities. However, we experienced large variances in prediction performances depending on the applied sampling strategy and hyper-parameters. Using the appropriate machine learning algorithm (DNN, KNN, Tree, Forest, or SVM) and suitable hyper-parameters, a prediction with an F-measure of 0.7 and above can be achieved. Nonetheless, there is a clear trade-off between precision and recall; over-sampling tends to improve recall, but decreases precision, while intensive under-sampling improves precision, but reduces recall significantly.

As our first results were promising, we decided to add hybrid source code metrics to the feature set. However, using our proposed vulnerability dataset was not feasible in this particular case (the reasons were explained earlier, in Section 4.1). To solve this problem, we used the BugsJS public dataset to find, extract, and map buggy functions in ESLint. We created three versions of a training dataset from the functions of the ESLint project. We ended up with a dataset containing 824 buggy and 1943 non-buggy functions with three sets of features: static metrics only, static metrics where the invocation metrics (NOI and NII) are replaced by their hybrid counterparts (HNOI and HNII), and static metrics with the addition of the hybrid metrics.

We trained nine different models in 108 configurations and compared their results. We found that using invocation metrics calculated by a hybrid code analysis as bug prediction features consistently improves the performance of the ML prediction models. Depending on the ML algorithm, applied hyper-parameters, and target measure we consider, hybrid invocation metrics bring a 2-10% increase in model performances (i.e., precision, recall, F-measure). Interestingly, even though replacing static NOI and NII metrics with their hybrid counterparts HNOI and HNII in itself improves model performances, most of the time, keeping them both yields the best results. It means that they hold somewhat complementary information to each other.

To achieve the highest accuracy and precision, one should choose the Random Forest Classifier method, but if the recall is also important and one wants to have an as balanced results as possible, the K Nearest Neighbors Classifier may be the best possible option. In the future, we plan to extend the set of predictors with various history and textual metrics, as well as other hybrid metrics in order to further improve the issue prediction at the level of JavaScript functions.

5

Studying Typical Security Issues and Their Mitigation in Open-Source Projects

5.1 Overview

Software security is one of the most striking problems of today's software systems. With the advent of low-cost mobile/IoT devices connected to the Internet, the problem of insecure applications has risen sharply. Large impact security vulnerabilities are explored on a daily basis, for example, a serious flaw [149] has been discovered in 'Sudo', a powerful utility used in macOS this February. Security problems can cause not just financial damage [19], but can compromise vital infrastructure, or can be used to threaten entire countries. Security issues can also be dangerous to humanity, for example, in a pandemic situation [118, 82, 87].

As security issues can mean a lot of things, we would like to note that in this chapter, we use CVEs as the definition of vulnerabilities. CVEs (short for Common Vulnerabilities and Exposures) [42] are publicly disclosed cyber-security vulnerabilities and exposures that are stored online and are freely browsable. These can be categorized into CWEs (short for Common Weakness Enumeration) [111], which is a widely adopted categorization for vulnerabilities.

Our focus in this chapter is to examine vulnerability mitigation (i.e. corrective code changes to resolve security vulnerabilities) within the open-source community, their typical types, and their other characteristics. By reacting to emerging security vulnerabilities an open-source community can contribute to building more secure solutions, as a lot of industrial applications rely on open-source libraries. By getting a detailed picture of what security vulnerabilities are mitigated and when in the open-source community of these languages, we can identify vulnerability categories that are not sufficiently addressed, explore patterns that might help to build more efficient vulnerability prediction models, or even discover some patterns that may help in generalizing the models. Moreover, we can create educational materials that can help individuals in becoming more prepared for vulnerabilities, we can emphasize the poorly addressed vulnerability categories. These and similar studies help developers to be more aware

of the security issues that might occur in the programming languages in which they work.

In Section 5.3, we used the rich dataset of the Software Heritage Graph Dataset [127]. We investigated the typical security vulnerabilities in the JavaScript and Python open-source communities, the mitigated issues' types, and how they relate to each other. We examined the speed of the JavaScript and Python communities, i.e. how quickly the communities mitigate a newly published security vulnerability. We found that the JavaScript projects refer to security vulnerabilities falling into 87 different categories, the Python projects to 71, out of which 55 security vulnerability categories are common. Despite the large intersection in the security vulnerability types, the number of mitigated vulnerabilities differ significantly depending on the language of the projects. For example, Cross-site Scripting (CWE-79), Path Traversal (CWE-22), Improper Input Validation (CWE-20) and Uncontrolled Resource Consumption (CWE-400) type of vulnerabilities are mitigated mostly in JavaScript projects, while Resource Management Errors (CWE-399) and Permissions, Privileges, and Access Controls (CWE-264) are mitigated mostly in Python. The growing number of vulnerability mitigating commits is a common tendency in both languages, but it is proportionate to the growth of the total number of commits. The vulnerability mitigation per total commit ratio increases only slowly, however, there was a significant increase in the amount of vulnerability mitigation in the year 2018 for both JavaScript and Python projects (see Figure 5.2). Regarding the number of days elapsing between the publish date of a particular security vulnerability and the date of the first commit with its mitigation is varies to a large extent. Typically, Python commits mitigate vulnerabilities no older than 100 days, while some JavaScript commits mitigate vulnerabilities older than a year.

Next to the Software Heritage Graph Dataset, which is a very valuable source of data, we also applied data mining techniques to collect security issues. Mining data on our own allows us to investigate interesting questions that were unfeasible using the Software Heritage Graph Dataset. We also created several tools to support researches like this one, the tools used are publicly available on GitHub.

In Section 5.4, we present the approach we used to gather data from GitHub, and the results of a small-scale, open-source study that aims to show the differences between programming languages, based on their activity when it comes to fixing security issues. We follow the basic ideas laid out by the work of Matt Bishop [30] with the design of our study approach. Our choice is to use data from as many languages as we can, including C, C++, BitBake, Go, Java, JavaScript, Python, Ruby, and Scheme programs, so we have the advantage of not constraining our field of view to only certain kinds of projects. And probably, the more languages we study, the more general results we obtain. For numerous programs written in these different languages we extracted and analyzed the type of vulnerabilities found and fixed in the programs, the time it took for the fix to occur, the number of people working on a given project while an issue was active, and the number of changes to the code and files required to eliminate the issue. The results show that while the severity of an issue may correlate with the time it takes to fix it, that is not the case in general. Averages show a similar pattern, which is likely because of the reintroduction of the same issues several times in larger projects. We found that smaller and more user interface-focused projects rarely document CVE fixes, however, larger-scale projects, especially those concerning backend solutions and operating systems (package managers, etc.) are more inclined to state major bug fixes.

We also found that in some projects, the developers prefer to only mention CVEs at larger milestones or releases, while in others, they were present in the exact commit they were fixed in. We also look at CWEs more specifically, their prevalence in different languages. Some of these are language-specific, while others are more general.

The chapter is organized as follows. We review works similar to ours in Section 5.2. Afterwards, we present our approach to mining data from the Software Heritage Graph Dataset in Section 5.3, as well as the results we obtained. In Section 5.4, we present our developed tools and the methodology we used to collect data from Git. We also present the results we could achieve from the collected data. We enlist the possible threats to our work in Section 5.5. Finally, we summarize the chapter in Section 5.6.

5.2 Related Work

There are plenty of previous works investigating different aspects of security vulnerabilities. By exploring the life-cycle of the vulnerabilities, one can understand their nature better, which helps to prevent, find, or predict them.

Li and Paxson [92] conducted a large-scale empirical study of security patches. They investigated more than 4,000 security patches that affected more than 3,000 vulnerabilities in 682 open-source software projects. They also used the National Vulnerability Database (NVD) as a basis, but they used external sources (for example GitHub) to collect information about security issues. We rely only on data provided by NVD [123] or MITRE [42, 111]. In their work, they investigated the life-cycle of both security and non-security patches, compared their impact on the code base, and their characteristics. They found out that security patches have a lower footprint in codebases than non-security fixes; a third of all security issues were introduced more than 3 years before the fixing patch, and there were also cases when a security patch failed to fix the corresponding security issue.

Frei et al. [59] presented a large-scale analysis of vulnerabilities, mostly concentrated on discovery, disclosure, exploit, and patch dates. The authors have found that until 2006, the hackers reacted faster to an exposed vulnerability than the vendors.

Similar to the previous work, Shahzad et al. [140] presented a large-scale study about various aspects of software vulnerabilities during their life cycle. They created a large software vulnerability dataset with more than 46,000 vulnerabilities. The authors also identified the most exploited forms of vulnerabilities (for example DoS, XSS). In our research, we also use categories, however, our categories are defined by CWE which is a widely used and accepted categorization. They found that since 2008, the vendors have become more agile in patching security issues. They also validated the fact the vendors are getting faster than the hackers since then. Moreover, the patching of vulnerabilities in closed-source software is faster than in open-source software.

Kuhn et al. [85] analyzed the vulnerability trends between 2008 and 2016. They also analyzed the severity of the vulnerabilities as well as the categories. Their study showed that the number of design-related vulnerabilities is growing while there are several other groups (for example CWE-89 (SQL Injection)) that show a decreasing trend.

In their work, Wang et al. [154] used Bayesian networks to categorize CVEs. They used the vulnerable product and CVSS [1] base metric scores as the observed variables.

[1] Common Vulnerability Scoring System, as presented by Mell et al. [106]

Although we do not use any machine learning methods in this study, our long-term goal is to use various machine learning methods using the data presented in this study. Wang et al. proved that categorizing CVEs is possible and machine learning can do that.

Gkortzis et al. [61] presented VulinOSS, a vulnerability dataset containing the vulnerable open-source project versions, the details about the vulnerabilities, and numerous metrics related to their development process (e.g. whether they have tests or numerous static code metrics).

In their work, Massacci et al. [103] analyzed several research problems in the field of vulnerability and security analysis, the corresponding empirical methods, and vulnerability prediction. They summarized the databases used by several studies and identified the most common features used by researchers. They also conducted an experiment in which they integrated several data sources on Mozilla Firefox. The authors also showed that different data sources might lead to different answers to a specific question. Therefore, the quality of the database is a key component. In our paper, we try our best to provide a usable, good quality database for further researches.

Abunadi et al. [15] presented an empirical study aiming to clarify how useful cross-project vulnerability prediction could be. They conducted their research on a publicly available dataset in the context of cross-project vulnerability prediction. In our research, we collected data from several programming languages. Hence we believe that our dataset can be useful in cross-project vulnerability prediction.

Xu et al. [159] presented a low-level (binary-level) patch analysis framework, that can identify security and non-security related patches by analyzing the binaries. Their framework can also detect patterns that help to find similar patches/vulnerabilities in the binaries. In contrast to their work, we use data mining and static process metrics. Therefore, our approach does not need any binaries, it does not require the project to be in an executable state, which can be extremely useful when a project's older version could not be compiled.

Vásquez et al. [153] analyzed 660 Android-related vulnerabilities and their corresponding patches. Their approach uses NVD and Google Android security bulletins to identify security issues. Despite the fact that we do not include Android security bulletins in this research, we plan to extend our scope in the future and include those vulnerabilities too, as our framework is extensible.

5.3 Exploring the Security Awareness of the Python and JavaScript Open Source Communities Using The Software Heritage Graph Dataset

In this section, we investigate the typical security issues and their mitigation in JavaScript and Python open-source communities. We used the Software Heritage Graph Dataset as our data source. The mission of Software Heritage is to collect, preserve and share all source code and its development history that can be found on the internet [43]. The dataset contains information about tens of millions of software projects, and it grows day by day.

5.3.1 The Software Heritage Graph Dataset

This dataset is a fully-deduplicated Merkle Distributed Acyclic Graph (DAG) [107] representation of the Software Heritage archive; it links together the source trees (directories and source code files), the Version Control System (VCS) commits and releases (tags), and some other meta information about crawling (when and where the data comes from). The content of the dataset comes from various platforms: source code hosting platforms (e.g., GitHub, GitLab.com); FOSS distributions (e.g., Debian); and package managers for specific programming languages (e.g., PyPI, npm).

We performed our study using the compressed PostgreSQL format [126] of the full Software Heritage Graph Dataset [127]. It took us several tries to correctly import the dataset into a local database. With some modifications to the original load script (e.g. removing concurrent index creation, as it is only justifiable in a production environment[2], but introduces a huge overhead), we managed to import the whole database into a local server.

The technical specifications of the database server we used were a 20-core Intel CPU (2,6 GHz), 90 Gbs of RAM, 5 Tb SSD. Despite the quite strong hardware, the data import and queries were rather slow due to the enormous size of the database. To speed up the process, we created intermediate tables from the relevant information in a filtered and transformed way.

5.3.2 Approach

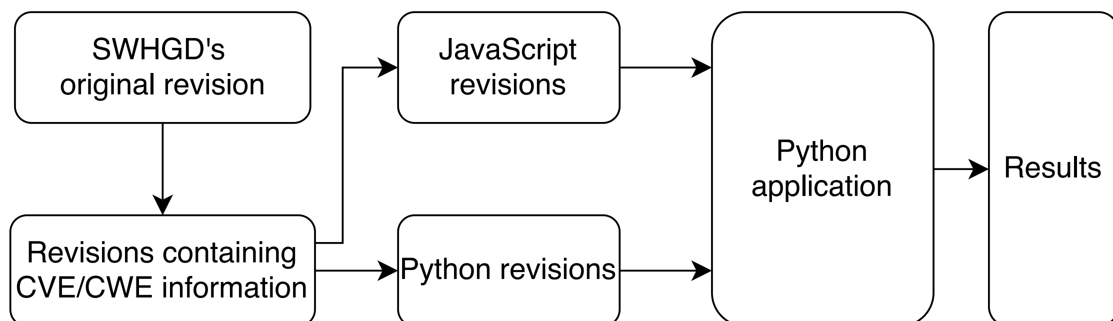


Figure 5.1. The schematic view of our approach

Our approach is based on collecting the vulnerability mitigation commits for JavaScript and Python projects from the dataset, which are potentially connected to a public CVE [42] entry. To achieve this, we used a very simple but effective heuristics-based approach, similar to those widely used in works related to bug data collection [68, 146].

Figure 5.1 depicts our overall approach. First, we searched for the commits containing the patterns “CVE-”, “CWE-”, “NVD-” (all of them are case insensitive) in their commit messages using SQL queries. Referring to a CVE or CWE identifier in the commit message is a widely used practice in case of vulnerability mitigation patches, so the community can understand why the given commit is extremely important and urgent to be merged. By filtering the `revision` table, we created a temporal table called `cve_revs` with 357,757 rows (from the original 1.26 billion rows).

[2]According to the official documentation: `https://www.postgresql.org/docs/11/sql-createindex.html`

After the first filtering step, we had to identify the programming language of the project a given commit belongs to. Since the structure of the database did not provide an effective way to do this, we used the information retrieved from the revisions' root directory:

- We considered a revision as a Python revision if its root directory contained either `__init.py__` or `setup.py`. Without at least one of these files, the project cannot be used as a Python module [134, 128, 160] (nor published on PyPI [133]), therefore, it is a viable heuristics to detect Python projects.
- We considered a revision to be a JavaScript revision if its root directory contained either `index.js`, `app.js`, or `server.js`, as one of these files will most likely be included in the root directory [122] of a JavaScript project. We did not consider `package.json` for identifying a revision as a JavaScript revision because `package.json` is often used in other languages as well, such as PHP (e.g., Symfony uses `package.json` to manage tools that are necessary for packing the application's frontend [151]).

Based on this second round of filtering, we got 3,718 rows for Python and 4,136 rows for JavaScript, which we stored in two new tables: `cve_revs_py` and `cve_revs_js`, respectively. We analyzed the data collected in these instead of the original `revision` table.

Tools and Queries for Data Mining

We processed the collected Python and JavaScript revisions using Python scripts and the pandas [104] framework, and used regular expressions[3] to find and extract the CVE/CWE identifiers from the commit messages. All the used regular expressions and extraction scripts for finding CVE/CWE and vulnerability mitigating revisions are available in our online asset package.[4] We also tried to filter commits for "NVD", but there were no matching commit messages. If a commit message contained more than one CVE or CWE reference, we extracted all and considered them separately (i.e. the commit contained mitigation for more than one vulnerability). Since a commit message can contain the same CVE/CWE IDs several times (for example, it can be in the first line of the commit message, and later it can appear in the description as well), we had to remove the duplicates. Thus one CVE/CWE entry is considered only once per revision.

Several rows have not been filtered out in the first step, but in the processing step we could not find any CVE/CWE IDs in their commit messages. We examined and validated all of these cases by hand. These revisions contained messages that could pass our first filtering but did not mention any valid CVE/CWE IDs, for example, *execve-safe, Glennvd-patch-1, nvd-downloader, no CVE-id.*

As we focused on the types of vulnerabilities which can be described by the CWE identifier of the security problem category the vulnerability belongs to, we had to link each CVE entry to the corresponding CWE category of the vulnerability. To achieve this, we relied on the data provided by the National Vulnerability Database [123]

[3] $(CVE - \backslash d\{4\} - \backslash d\{4,\})$, $(CWE - [\backslash d]\{1,4\})$, and $(NVD\ .+)$

[4] `https://doi.org/10.5281/zenodo.3699486`

and used a customized version of CVE manager by Atlasis [5] to parse the JSON data files describing the CVE entries with meta-information, like its corresponding CWE category. Besides the CWE group of a CVE entry, we also extracted the publishing date, severity, and the base impact score of every CVE entry.

Some revisions contained references to CWE groups without mentioning any specific CVE entries. These revisions were mapped directly to the referenced CWE categories.

5.3.3 Results

After all filtering steps, we identified a total number of 3,458 vulnerability mitigation commits (i.e. commit messages containing valid CVE or CWE identifiers) for JavaScript and 2,884 for Python to which we were able to resolve the corresponding CWE category as well. Figure 5.2 shows the ratio of commits over the years in terms of the average number of vulnerability mitigation commits per 100k commits.

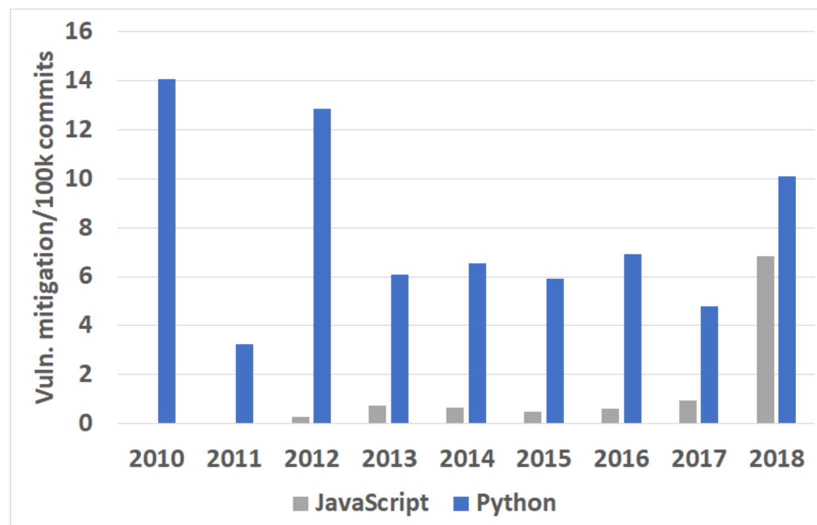


Figure 5.2. Vulnerability mitigation ratio per year

While the Python vulnerability mitigation ratio is quite stable, the same ratio for JavaScript projects grows consistently from 2015, with a large peak in 2018, but is still lower than that of Python projects. As there are no JavaScript commits in the Software Heritage Dataset before 2010, and the data for 2019 is still incomplete, we omitted those years from the analysis.

Table 5.1 provides further details on the number of detected vulnerability mitigation commits and the total number of commits in the analyzed years. The distribution of the referenced CWE vulnerability types is depicted in Figure 5.3.

Typical Security Issue Types

We examined the extracted vulnerability mitigation commits with 103 different CWE categories. From these 103, 55 CWE types occurred in both JavaScript and Python commit messages, while 32 CWE groups were found only in JavaScript projects, and

[5] Our version is available on `https://github.com/gaborantal/cve_manager`, while the original one can be found on `https://github.com/aatlasis/cve_manager`

Table 5.1. Commit statistics per year

Year	Vuln. JS	Vuln. PY	Total JS	Total PY
2010	0	225	102,525	1,597,160
2011	0	67	675,492	2,068,155
2012	6	343	2,078,887	2,663,836
2013	41	209	5,705,696	3,436,804
2014	84	291	12,692,836	4,440,660
2015	111	328	23,794,463	5,537,294
2016	239	453	38,990,699	6,527,350
2017	393	329	40,883,417	6,835,803
2018	2584	639	37,729,971	6,315,866

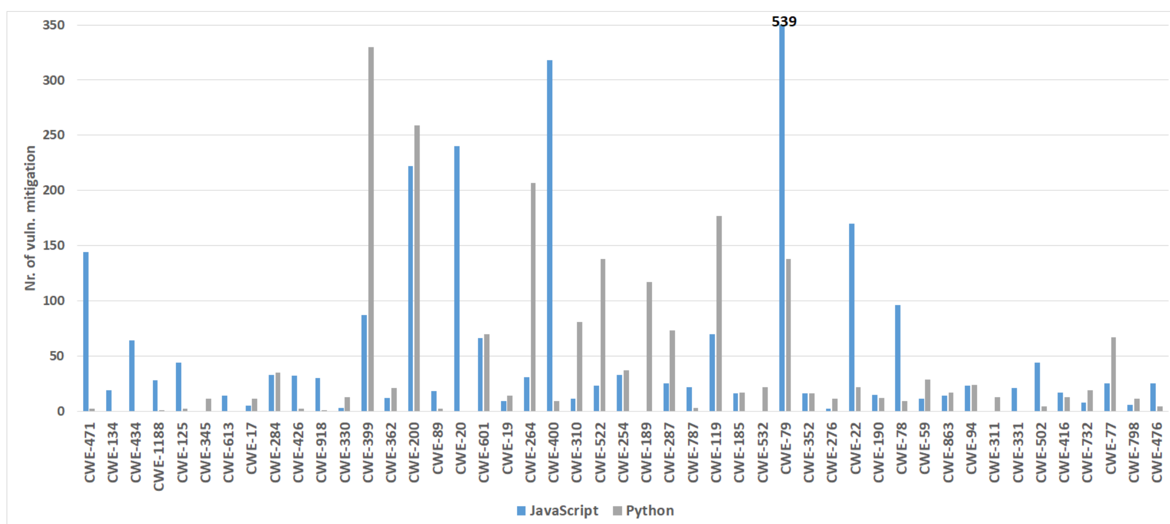


Figure 5.3. Number of security issues found with the given CWE types

16 only in Python projects (however, the number of vulnerabilities with such types were very low).

We examined the most popular CWE categories in more detail. The CWEs having at least 150 references in either of the analyzed languages are described in Table 5.2.

Interestingly, except for CWE-200, which is the type of vulnerability mitigated in more than 200 commits in both languages, each of the other six CWE groups can be attributed to either JavaScript or Python projects (i.e. one of the languages contain the majority of the mitigation to these vulnerability types). On the one hand, Cross-site Scripting (CWE-79), Path Traversal (CWE-22), Improper Input Validation (CWE-20), and Uncontrolled Resource Consumption (CWE-400) type vulnerabilities are mitigated mostly in JavaScript projects. All these vulnerability types are primarily relevant for web applications, where JavaScript is heavily used at the client-side, thus it is more probable that a JavaScript project encounters such vulnerabilities. On the other hand, mitigation of Resource Management Errors (CWE-399), Permissions, Privileges, and Access Controls (CWE-264), and Improper Restriction of Operations within the Bounds of a Memory Buffer (CWE-119) type vulnerabilities occur in Python commits mostly. These are more relevant at the server-side, where Python seems to dominate. There is a significant overlap in these categories as well, so projects from both languages

have vulnerabilities with all these CWE types, but based on the data we have, it seems that these are more typical for a particular language.

Table 5.2. Most referenced CWE categories and their description

CWE Identifier	Description
CWE-79	Improper Neutralization of Input During Web Page Generation (Cross-site Scripting).
CWE-399	Resource Management Errors.
CWE-200	Information Exposure.
CWE-20	Improper Input Validation.
CWE-264	Permissions, Privileges, and Access Controls.
CWE-400	Uncontrolled Resource Consumption.
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer.
CWE-22	Improper Limitation of a Path-name to a Restricted Directory (Path Traversal).

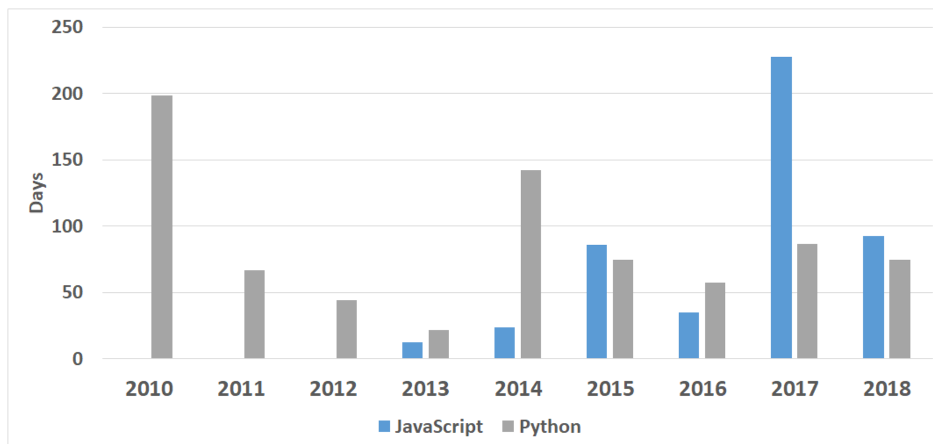


Figure 5.4. Average number of days between mitigation commit date and CVE publish date grouped by years

Reaction Times to Security Issues

We analyzed the average number of days elapsing between a mitigation commit date and the publish date of a CVE entry mentioned in that commit. Figure 5.4 depicts a general overview of these average number of days per year. We can see that it takes about 100 days on average for both communities to start mitigating a public vulnerability in their code-base, with some peaks in years 2010 and 2014 for Python and 2017 for JavaScript. Therefore, we can conclude that at a very general level, neither the JavaScript nor the Python communities react fast to appearing vulnerabilities in their code. It would also be interesting to see if there are reported CVE entries that are never mitigated in reality, but it would require an entirely different methodology and could be good for future research.

We also examined the eight most prevalent CWE categories from the same aspect. The average number of days elapsed between the publish date of a CVE entry and

the date of its mitigation commit for the top eight CWEs are shown in Figure 5.5. The Python community reacts $1.5 - 14$ times faster to these types of vulnerabilities than the JavaScript community; most of the mitigation commits appear 50 days or less after the publish date of the corresponding vulnerabilities. In the case of JavaScript, only vulnerabilities from three CWE categories enjoy extra care (CWE-20, CWE-200, CWE-400), all the others are mitigated after at least 100 days.

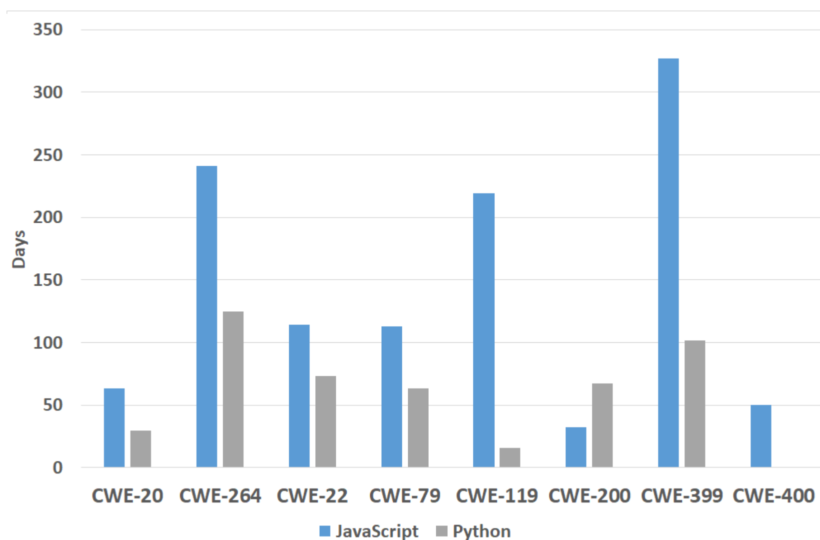


Figure 5.5. Average number of days between commit date and issue publish date for the most common CWEs

The JavaScript community reacts exceptionally fast to information exposure (CWE-200) type vulnerabilities (after only 32.5 days on average), while improper input validation (CWE-20) and uncontrolled resource consumption (CWE-400) are mitigated after about 50 days on average. Interestingly, the vulnerabilities falling into the CWE categories characteristic to Python (CWE-264, CWE-399, and CWE-119) are mitigated after 200 days or more.

5.4 A Data-Mining Based Study of Security Vulnerability Types and their Mitigation in Different Languages

Despite the Software Heritage Graph Dataset offering a huge amount of data, a lot of information is missing, which we could benefit from in this particular case. To overcome this, we designed our own data mining approach that – besides mining the data – calculates several other, useful metrics, such as the line and file changes, or the number of commits during the fixing of a CVE (the latter one was not used in this study).

In this section, we present the mining approach we used to gather and process data from several GitHub projects. First, we present our tools that define our approach, then we present the results we obtained.

5.4.1 Approach

We created 3 separate tools that communicate with each other, although they are designed and implemented to be usable as standalone tools as well. As a preliminary step, we needed hundreds of input projects, so we used GHTorrent [63] to acquire repositories that we can mine.

The overview of our approach is depicted in Figure 5.6. The approach we took can be best explained through the tools we created to collect the necessary information. We will use the described tools as bullet points to illustrate the flow of the entire study and the inner workings of the miner program.

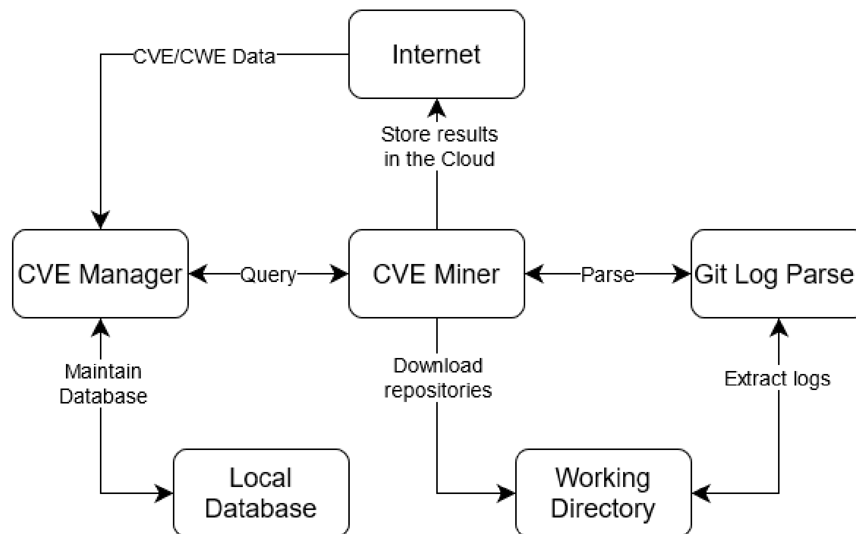


Figure 5.6. A schematic representation of our mining process

CVE Manager

CVE Manager[6] is the backbone of most statistics and is essential to validating the found CVE entries. It is a lightweight solution that downloads the CVE data from the MITRE Corporation's website [7]. We store most of the collected data in a PostgreSQL[8] database. The tool is used to query for CVE entries found by the miner and some of their properties like their *id, impact score, severity.*

Git Log Parser

The other important tool used by our miner tool is our Git log parser[9] solution. It simulates user commands using the Python `subprocess` module, so that some of GitHub's API limitations are bypassed. However, our solution requires the download of the source code repository (with `git clone` command, as the script is prepared to mine local directories for data. The parser first navigates to the path provided by the user (through command line input), then issues the `git log` command that lists every commit and their meta-data. It then saves this information into a list that will later be

[6] `https://github.com/gaborantal/cve_manager`
[7] https://www.mitre.org/
[8] https://www.postgresql.org/about/
[9] `https://github.com/gaborantal/git-log-parser`

written into a JSON file. This basic data is being extended with the line and file change information by comparing each commit to its predecessor with the git diff command.

The reports generated by the Git log parser can be useful in a variety of situations similar to ours, where an external utility needs the logs of a specific git repository. Some of its results are not used by the miner but are intended for later use, for example, the parser could check whether a commit is a merge, which is currently ignored in finding CVEs.

CVE Miner

The main tool is the CVE Miner[10], which uses both the CVE Manager and the Git log parser to create a JSON file and a database entry for each CVE found, and presumably fixed in the actual repository. Figure 5.6 represents the inner working of the miner and its interaction with the other tools.

The miner requires some initial setup since the CVE data needs to be downloaded and inserted into a local PostgreSQL database. This is done in two steps. In the first step, the data is collected into a local directory (called *NVD*) from which we can read. Then, the read data is inserted into the database.

There are multiple ways to start working with the CVE Miner. It can mine from both local and online sources. These options can be accessed using the command line interface. When an online source is provided, a directory called "repos" will be created if it does not already exist, and the given repository will be automatically cloned into it. Then the CVE Miner will continue as if a local directory had been provided. Multiple targets can be specified at once using a JSON file and the appropriate command line argument.

Afterwards, the miner processes the repositories by using the Git Log Parser tool. After the JSON file is generated, the tool searches for CVE entries in the commits' messages. If a CVE is mentioned once, the miner assumes that the associated commit fixes the CVE. If it is mentioned multiple times, it is assumed that the first occurrence implies that the CVE is found in the code, and any subsequent mentions are the fixes for that vulnerability. During this process, other data is collected, including but not limited to the contributors, the number of changed files, and the number of commits between the finding and fixing of the CVE.

The next step is the calculation of statistics. The miner uses the previously acquired information to calculate the average time between the commit that found the CVE and the commit that fixed it. The tool also calculates the correlation between a CVE entry's severity and the time needed to fix it.

The last step is to store data. By default, the miner creates a JSON file containing all the found CVEs and the calculated statistics. If chosen, the tool also uploads it to an Airtable[11] database. We implemented the database upload as a convenience feature, so that multiple miners can run on multiple computers, while the stored data is being collected into one single database. Other databases could easily be supported.

Approach Summary

Our main point of interest during this study was the collection of security-related data, thus a large emphasis has been put on it. We focused mainly on creating useful utilities

[10] `https://github.com/gaborantal/cve-miner`
[11] https://airtable.com/product

for later research, and we trust we succeeded when it comes to most of the tools created.

We took the approach of looking only for mentions of the text "CVE" in commit logs as it is a fast solution providing sufficiently good approximation. The best way to improve current data is of course to collect a much larger amount of it.

5.4.2 Results

In this section, we present our findings. First of all, we needed to select the projects we want to include in this study. It is mentioned in Section 5.4.1 that we used GHTorrent to collect repositories. From hundreds of repositories, we manually picked several to test whether they contain any CVEs at all. If a project did not contain any CVEs, we dropped it from our list.

Due to our resource and time limitations, we included 5 projects from each language. Our main constraint was to choose both smaller projects with fewer developers, as well as larger projects with larger developer communities. Finally, we managed to select 5 projects from each language, in different sizes, and with different purposes.

Time-Based Statistics

Time elapsed between the finding and fixing commit. These statistics can be interpreted in several ways. First, we address the intended purpose, showing how much time it takes to fix a CVE entry on average. This is more accurate on projects on a smaller scale with shorter lifespans since those have a lower chance of having false fixing claims and reoccurring issues.

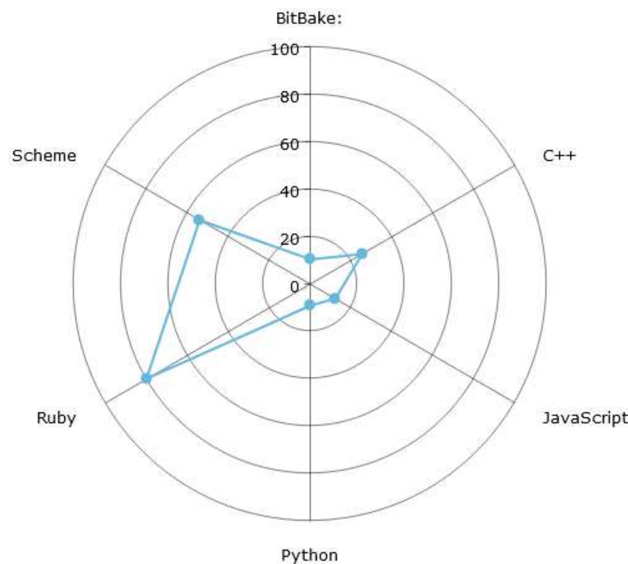


Figure 5.7. The average time elapsed between finding and fixing a CVE (in days)

The second way the statistics can be interpreted is, as we mentioned previously, an indication of reoccurring vulnerabilities. Most of the time a CVE entry is mentioned in the context where it is claimed to be fixed, which is not surprising since one does not want to disclose an actual vulnerability in their program before fixing it. Based on this fact, most of the CVEs should only be mentioned once, when they are getting fixed. However, this is not the case in most large scale projects. We assume that this happens because later changes may reintroduce a previously fixed vulnerability, which

is likely because in larger systems it is a lot harder to foresee every possible outcome a change might cause. During our manual inspection, we experienced that projects with a longer code history usually have more reoccurring issues than others.

When it comes to languages, a similar pattern can be observed (as can be seen in Figure 5.7). The differences are drastic since the scale and age of the analyzed projects vary. Most of the C++ and Scheme projects we looked at were quite large projects, hence the reason for their dominance in the chart. Ruby is an outlier, where it is common that an already fixed vulnerability resurfaces years after the initial fix. The other reason vulnerabilities in some languages are more prevalent than in the others also has to do with the fact that larger systems usually do not allow developers to make changes directly to the working tree; merges that happen later can also increase this fixing time. This is not an issue since an error being fixed in a branch should not be considered fixed in the application until it has been merged.

Time elapsed between the publication and fixing of a CVE. The nature of this result is similar to the previous one, however, it also takes into account the time each CVE spent in the code unnoticed after its publication. The results are depicted in Figure 5.8. Most of the languages show similar attributes compared to the previous chart, however, when it comes to BitBake and Scheme, a clear bump is visible, implying that it takes longer to come up with the first fix for an issue in programs written in BitBake and Scheme. We would like to note that BitBake and Scheme are not amongst the most popular choices nowadays, according to TIOBE Index[12]. This might also cause delays in fixing vulnerabilities, as their developer communities are quite small.

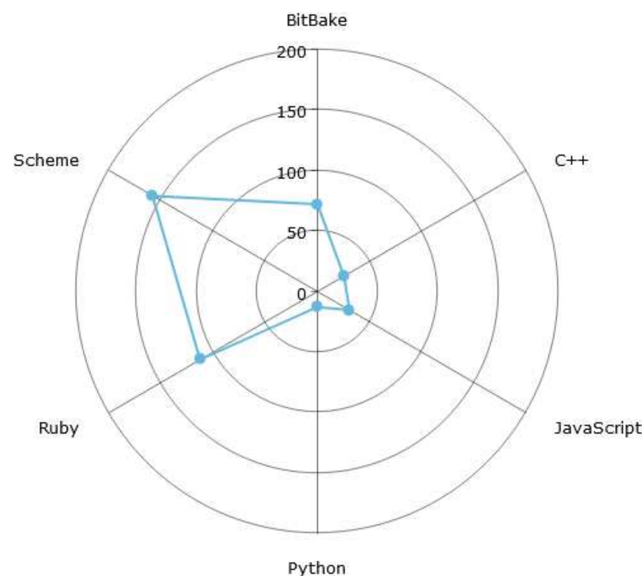


Figure 5.8. The average time elapsed between the publication and fixing of a CVE (in days)

Correlation between time and severity. Figure 5.9 shows the correlation between the aforementioned statistics and the severity of CVEs. The correlation between the fixing and the finding of a CVE can be attributed to the difficulty of the issue at hand, and the thoroughness of the testing.

[12]`https://www.tiobe.com/tiobe-index/`

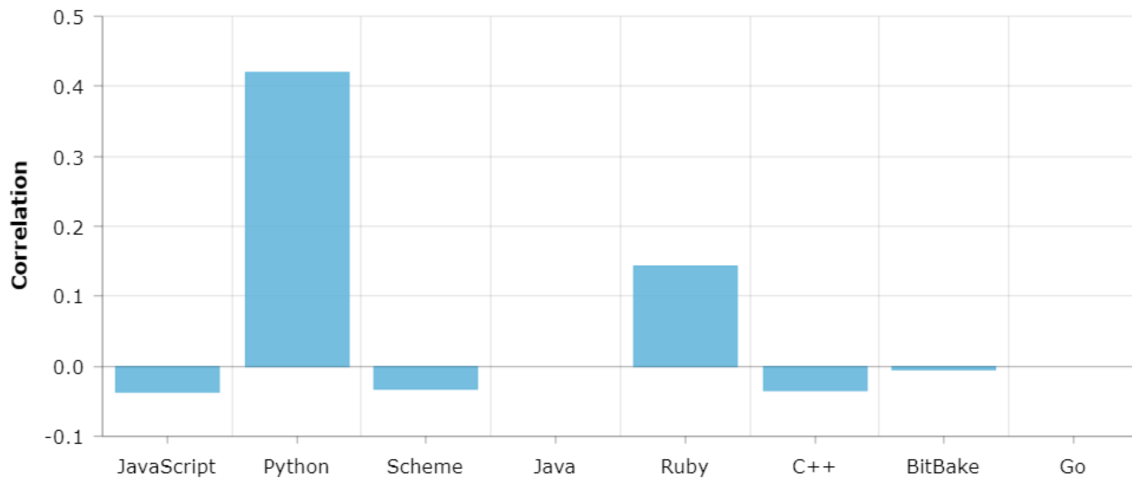


Figure 5.9. The correlation between the base score (severity) and time taken fixing CVEs

The correlation between the publication date of CVEs and the time it took to fix them shows how prepared developers were when it came to fixing these vulnerabilities. For example, in the case of Python, the more severe problems were solved quicker than the other, less severe issues. This might imply that they put a larger emphasis on getting rid of the more severe issues.

Activity-Based Statistics

Active contributors and commit count during the fixing of a CVE. The results in Figure 5.10 and Figure 5.11 showcase not only how quickly some issues might get fixed, but the activity within the project during the process of fixing an issue. Both charts show activities within the projects. Figure 5.10 presents the number of contributors who were working on the project between the first and last commit mentioning the same CVE. As we can see several tens of contributors (e.g. JavaScript, Scheme) or even more than 100 (Ruby) contributors might work on a codebase in the period of fixing a single vulnerability.

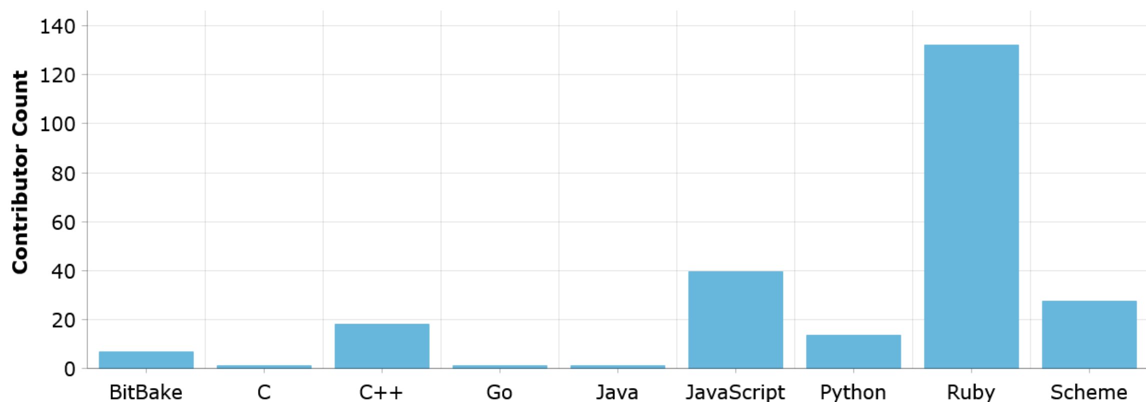


Figure 5.10. The average number of contributors between the finding and fixing commit

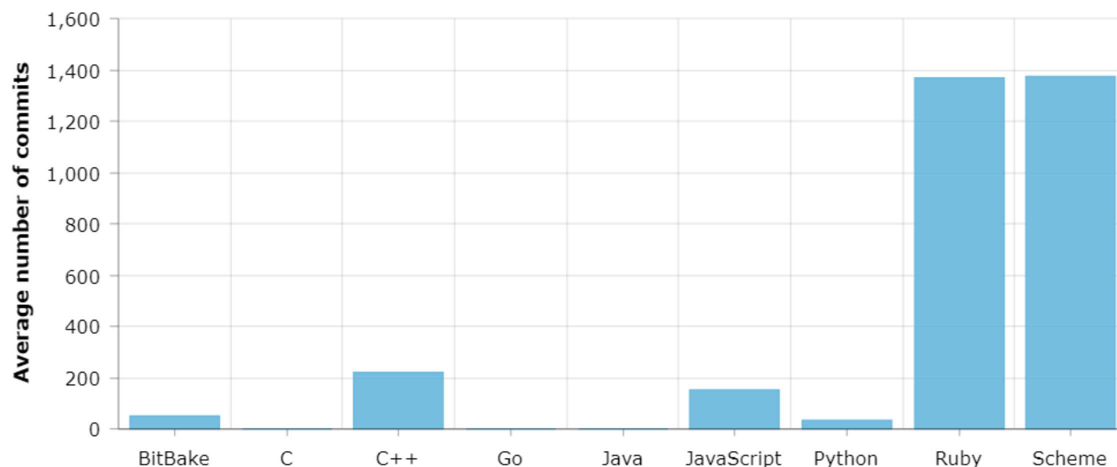


Figure 5.11. The average number of commits between the finding and fixing of a CVE

Table 5.3. Average total code changes

Language	Average total lines changed	Average total files changed
BitBake	153.25	2.5
Go	123.23	5.69
JavaScript	288.46	4.53
Python	84.16	5.7
Ruby	58.77	4.19

The number of commits in the vulnerability fixing period is depicted in Figure 5.11. We can see that the highest number of commits during the issue fixing period happens in Ruby and Scheme (almost 1400) projects. This implicates that a lot of code changes happen until a security vulnerability is finally fixed.

Average File and Line Changes

The average changes to files and lines show how impactful an average CVE is in each language. These numbers are of course extremely varied, not only per language but per project as well, since some might use fewer but longer files to store the same code, while others might separate code a bit more. They also might only mention CVEs at larger milestones or merges, making some of the results disorderly high. Table 5.3 shows the average total line and file changes per language upon fixing a CVE.

Most Common CWEs by Language

The usefulness of CWEs. CWEs are a grouping used for CVEs based on the type of weakness they cause. Knowing which CWE is most common in a language can be extremely useful when it comes to finding, fixing, and looking out for problems. This information is also helpful when it comes to trainings, so that companies can focus on specific areas (of a given language, or a framework) in order to prevent the vulnerabilities from occurring. This can reduce the time and energy needed to overcome

certain vulnerabilities, and can improve the code quality.

The data presented in Table 5.4 is of course not entirely indicative of each language as our scope is very limited, however, it might still be helpful, and give ideas of what types of security issues should be in focus.

Table 5.4. Most common CWEs per languages

Language	CWE	Percentage
BitBake	CWE-119	21.40%
	CWE-125	18.87%
	CWE-20	11.74%
C	CWE-20	50.00%
	CWE-400	25.00%
	CWE-125	25.00%
C++	CWE-119	92.39%
	CWE-200	5.71%
Java	CWE-502	45.00%
	CWE-20	20.00%
	CWE-200	15.00%
JavaScript	CWE-400	15.05%
	CWE-20	13.98%
	CWE-200	12.90%
	CWE-79	7.53%
	CWE-119	6.45%
	Other[13]	5.38%

Language	CWE	Percentage
Go	CWE-400	25.00%
Python	CWE-20	16.13%
	CWE-79	16.13%
	CWE-89	9.68%
	CWE-200	9.68%
	CWE-601	9.68%
	CWE-185	6.45%
Ruby	CWE-79	26.92%
	CWE-20	15.38%
	CWE-264	11.54%
	CWE-89	9.62%
	CWE-22	5.77%
	CWE-200	5.77%
Scheme	CWE-119	23.49%
	CWE-20	8.29%
	CWE-125	8.09%
	CWE-416	7.70%

As an example, the most common CWE in C++ is CWE-119 [14] which is basically the category for incorrect memory management. An example of a more generic CWE is CWE-20 [15], a possible cause of this is an improper input validation in the code.

For some of the languages with a more diverse set of CWEs, we created Figure 5.12 to visually illustrate the distribution of the most common CWEs.

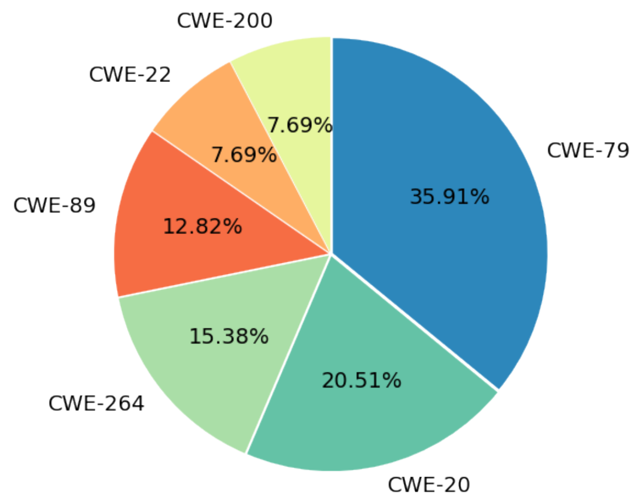
5.5 Threats to Validity

The main weakness of our results is the limited scale at which we operated. In Section 5.3, we used the Software Heritage Graph Dataset, but we only searched for Python and JavaScript projects. In Section 5.4, we only had the resources to mine a few repositories for most languages. However, the obtained results were similar to each

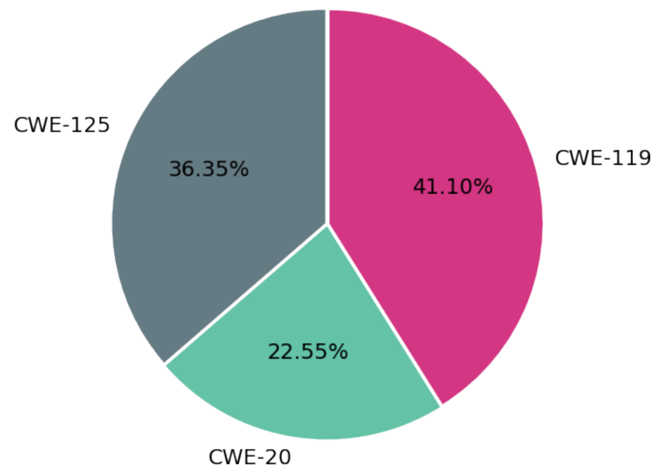
[13] NVD-CWE-Other: the CVEs falling into this category have a non-specified category. However, it can also happen that the actual CWE group is mentioned in the textual description of a given CVE (e.g., CVE-2007-0838). We only consider the type that has been set, the descriptions are not processed.

[14] `https://cwe.mitre.org/data/definitions/119.html`

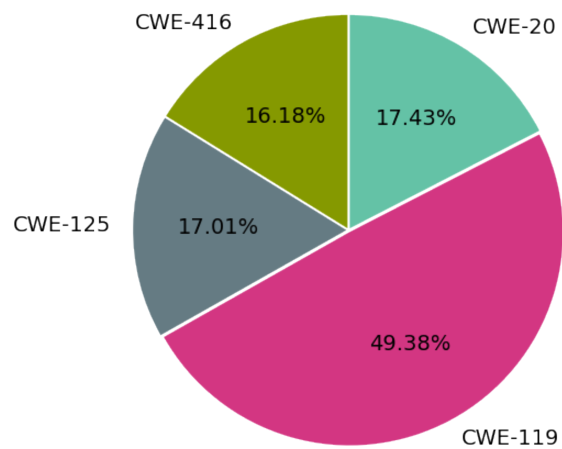
[15] `https://cwe.mitre.org/data/definitions/20.html`



(a) Ruby



(b) BitBake



(c) Scheme

Figure 5.12. Most commons CWEs for Ruby, BitBake, Scheme

other (in the case of Python and JavaScript). Hence we believe that the scale at which we operated did not affect the main results.

In Section 5.3, we had to apply heuristics to determine the language of the projects (as the exact solution would have been practically infeasible due to the database structure). Due to this, we might have omitted some projects, and we might have also identified some projects incorrectly. However, as our heuristics are based on widely established guidelines and best practices that most of the projects follow, the number of these projects should be minimal.

In Section 5.4, we developed tools to help collect and process data from Git repositories. The programs we created might contain bugs. We tried to mitigate this problem by a thorough code review of our developed tools. We also created test suites for all of our programs that run after every push, so that we can be certain that our program's output is never changed unintentionally.

One other major issue stems from the fact that we do not look at the code, but rely on the commit messages left by the developers. This can be troublesome when it comes to claiming that issues reappear, since it could be the case that they were never fixed in the first place. We assume that the last commit at which a CVE is mentioned is the last time it occurred, and has therefore been fixed. This might not be the case; it is possible that a fix happened later, but the developer forgot to mention it.

In most of the cases, the committers mention CVE identifiers explicitly, however, there are unusual references, for example "Fixed XSS (with CVE number 2020-100)" or "CVE-2020-20500/330/34/345". Also, there is a chance that a committer mentions a CVE in a context that is not related to fixing its underlying security issue. In such cases, we might drop valid vulnerability mitigation commits or include invalid ones. To estimate the impact of this threat, we manually evaluated more than 800 randomly selected commit messages from the identified revisions. In the vast majority of the evaluated cases, the commit messages refer to CVE IDs as we anticipated, thus the impact of this threat should be minimal.

We also do not account for merges, which can increase the number of lines needed for a fix. We believe that an issue is not fixed until it is merged into the master branch. However, if we count the lines in the commit that fixed the issue (commit fix_a), and we also count the same number of lines in a merge commit that merges commit fix_a among others, then the same commit gets taken into account twice, which might not be indicative of the actual amount of work needed for the solution. In these cases, we currently just count the lines twice, but this has caused some statistics to be left out since they portrayed false information because of the practices the developers used when they merged larger pieces of code at once.

5.6 Summary

In this chapter, we focused on analyzing security issues for several programming languages, using an already existing dataset, and a dataset we created. We also tried to determine if there are vulnerability type characteristics of languages.

Using the Software Heritage Graph Dataset, we analyzed the vulnerability mitigation commits in Python and JavaScript projects from two aspects. On the one hand, we identified the types of vulnerabilities (in terms of CWE groups) referred to in commit messages and compared their numbers within the two communities. The percentage of vulnerability mitigation commits compared to the total number of commits in projects

shows a growing tendency (sharper in the case of JavaScript, slower for Python). We detected 103 different CWE groups out of which 55 appeared in the projects of both languages. From the eight most prevalent vulnerability types, one was mitigated by both communities in equal numbers (CWE-200), but four (CWE-20, CWE-22, CWE-79, CWE-400) was typical to JavaScript, while three (CWE-399, CWE-264, CWE-119) to Python projects.

On the other hand, we examined the average time elapsing between the publish date of a vulnerability and the date of the commit mitigating it. We found that in general, neither the JavaScript nor the Python community reacts very fast to appearing vulnerabilities (i.e. it takes more than 100 days on average to mitigate a vulnerability after its publish date). However, this reaction is 1.5-14 times faster in the Python community for the most common CWE categories (even to the ones more typical to JavaScript projects), while the JavaScript community only seems to take a special interest in three CWE categories: CWE-200, CWE-20, and CWE-400.

In Section 5.4 we presented our approach for collecting data to this and similar studies; we also implemented several useful tools available on GitHub[16]. Using our tools, we investigated several vulnerabilities for programming languages that are currently popular, or were once widely used. We found that even at smaller sample sizes, specific weaknesses showed a clear trend in most of the languages we studied. For example, in the case of C++, CWE-119 (memory handling problems) was the most common issue type that developers faced. This may not surprise those familiar with the language, but for a new developer, it can be informative and show them what to pay attention to. A great example of how interesting these findings truly are is Ruby. It is visible that for Ruby developers, the biggest issue is CWE-79, improper neutralization of inputs [17]. These issues take less effort to fix than others, requiring on average about 60 lines of code and 4 file changes, however, the same issue might reappear later, as shown in Figure 5.7. It is also visible that while in Ruby it takes the least amount of lines to fix an issue, more severe vulnerabilities take longer to get rid of, as seen in Figure 5.9.

In conclusion, each language has its share of common weaknesses, which depend on a variety of factors, and being cautious of these is important. We also experience that fixing already published issues takes a great amount of time, even if an issue has already been noticed; and issues can reappear as time passes.

[16]`https://cveminer.github.io`

[17]https://cwe.mitre.org/data/definitions/79.html

"Talent is cheaper than table salt. What separates the talented individual from the successful one is a lot of hard work."

— Stephen King

6 Conclusions

In this thesis, we covered four topics and more than 6 years of research work. The covered topics include supporting C++ legacy compilation environments while enabling developers to use newer language standards, revealing the differences between static JavaScript call graph algorithms, building bug prediction models to predict software issues in JavaScript functions, and last, but not least, studying the typical security issue types in several programming languages.

First, we created a solution for *supporting legacy compilation environments* in C++ projects, meaning that our tool transforms the code that contains a subset of the new language features defined in C++11, to a functionally equivalent code that can be compiled with any standard C++03 compiler. We also created a test suite, and tested our tool on 6 real-world applications. Our results showed that the transformation framework is capable of transforming projects containing millions of lines of code.

In the field of *static JavaScript call graph algorithms*, we presented a systematic comparison of 5 state-of-the-art tools, using both a JavaScript benchmark and several real-world Node.js modules. We revealed both the similarities and the differences among the tools, and found that we cannot declare an absolute winner, as each tool has its strengths and weaknesses. We also showed that the combination of various tools yields the best results.

In *software issue prediction*, we presented a comparison of 8 well-known machine learning algorithms on predicting vulnerable JavaScript functions, using a newly created dataset that contains several static source code metrics on function level. As the results were encouraging, we widened our scope, and extended the feature set with two hybrid call graph based metrics, Hybrid Number of Outgoing Invocations (HNOI), and Hybrid Number of Incoming Invocations (HNII), which besides static call edges also use dynamic (run-time) function invocation information. We also created a hybrid call graph framework to ease future researches with hybrid analysis. We did a comparison of 8 well-known machine learning algorithms on predicting software bugs in JavaScript functions. We revealed that hybrid invocation-based metrics consistently improve the performance of the prediction models; depending on the machine learning algorithms, 2-10% increase in model performances (i.e., precision, recall, F-measure)

can be achieved.

Finally, in the field of *studying typical security issue types*, we presented our approach on how to collect the required data from the Software Heritage Graph Dataset, how one can mine data from any Git repository effectively. We created tools and a database to help researchers in this field. Our results revealed that there are typical vulnerability types in programming languages, and the mitigation process takes a lot more time than we would think. However, as time goes by, the vulnerability fixing process is getting faster and faster, which is reassuring.

Future Work

In spite of the achieved results, there are several opportunities for future work in all the topics we mentioned. *Supporting legacy environments* has its potential; extending our tool (both to support the C++11 as much as possible, and to support features from the newer language standards (defined in C++17/C++20)), or using our approach in other languages (for example, Java) could be very useful.

Our *comparative study on static JavaScript call graph* algorithms inspired others to do studies similar to ours [161, 75, 86], to understand the available tools better. We are currently working on an extension to our study, involving dynamic call graph construction algorithms, so that we will have a better understanding of how accurate static call graph tools are; and of the trade-off between using static or dynamic call graph builders.

Understanding the typical security issues and their mitigation more in different programming languages can help developers in many ways: we can create educational materials that can help individuals in becoming more prepared for vulnerabilities; developers will be more aware of the security issues that might occur in the programming languages in which they work. In addition, knowing the characteristics of the typical security issues in a language helps researchers fine-tune their tools and their prediction models. We also use the gathered knowledge to build more accurate vulnerability prediction models. Our plan is to include as many projects as we can, in order to have a bigger and better picture of the security issues in a given language.

The field of *predicting software issues* based on static and/or dynamic analysis has countless opportunities, especially nowadays. Combining software metrics with developer metrics or with process metrics has a lot of potential. We could use the combination of all available metrics to further enhance prediction. As of today, most of the prediction-related work uses class/file granularity. Quite a few studies deal with finer granulation, most of them use function-level prediction. Using our presented methodology to predict issues in other languages (e.g. Python, Ruby) could be very useful. We plan to extend our methodology to other dynamic languages. Furthermore, a prediction model that can predict vulnerable source code lines with a low false positive ratio would greatly facilitate the practical application of such prediction models. We plan to create a line-based prediction model that can accurately predict issues before they find their way into the live version of a software. Our preliminary results [113] are promising, however, there is a huge amount of work left to do in this field.

Appendices

Summary in English

After the outbreak of the new coronavirus, the number of daily cyber-attacks grew by 300%, causing more than 4,000 attacks a day [72]. The cyber-criminals are targeting the whole populace, including hospitals, researchers, and people who work from home [82, 87].

In this thesis, we focus on topics that emphasize the importance of software quality and security. First, we propose a solution to support elements defined in C++11 standard in a C++03 environment to enable developers to write less error-prone, more expressive code. Then, we present our comparative study on static JavaScript call graphs that demonstrates the similarities and differences between the most popular call graph construction algorithms. As we assumed, for a highly dynamic language such as JavaScript, there can be huge differences between the created call graphs. Next, we present a comparison of 8 well-known machine learning algorithms on predicting security issues in JavaScript functions. As a preliminary step, we only used static source code metrics as features. As the results were promising, we extended the scope of the research: we enhanced our model with invocation metrics coming from a hybrid (both static and dynamic) analysis based call graph. Finally, we show a study on typical security issues and their mitigation in open-source projects. The valuable information we gathered in this topic helps researchers fine-tune their machine learning models, and helps developers to know the possible vulnerabilities in their code more. The developed tools can help lead developers choose between open-source libraries.

The results are grouped into four major thesis points. Table A.1 shows the relation between the supporting publications and the thesis points.

I. Transforming C++11 Code to C++03 to Support Legacy Compilation Environments

The contributions of this thesis point are discussed in Chapter 2.

Newer technologies (e.g. programming languages, environments, libraries) change rapidly. However, various internal and external constraints often prevent projects (and teams) from quickly adapting to these changes. Keeping up to date with the newer technologies makes the software less error-prone and its performance

better, as more and more useful functions and features are being introduced in each and every change set. Despite this, customers may require specific platform compatibility from a software vendor, for example.

This thesis point deals with such an issue in the context of the C++ programming language. An industrial partner of the Department of Software Engineering of the University of Szeged is required to use Software Development Kits (SDKs) that only support older C++ language editions. They, however, would like to allow their developers to use newer language constructs in their code, and of course, developers are eager to use elements defined in newer standards of C++.

To address this problem, first of all, we did a thorough research, and compared all of the possible solutions that could be used in this particular case. We found that using LLVM and clang was the best possible way to implement such a tool. We designed and implemented a source code transformation framework to automatically backport source code written according to the C++11 standard to its functionally equivalent C++03 variant. With our framework, developers are free to exploit a large portion of the latest language features, while the production code (which is transformed with our framework) is still built by using a restricted set of available language constructs, thus making it compilable with a standard C++03 compiler. The tool is designed in such a way that it supports multiple transformations in one run; it runs incrementally (only the changed files are being transformed), we run the transformations in a specific order, so that multiple transformation can run in parallel. The transformations we implemented in our frameworks are: In-class data member initialization; Auto type deduction; Lambda functions; Attributes; Final and override modifiers; Range-based for loop; Constructor delegation; Type aliases; *Other transformations with limited functionality.* We also evaluated our framework on 4 open-source, and 2 closed-source industrial projects. We reported the tool's performance with different parallelization settings.

Our solution is open-source, and available on GitHub: `https://github.com/sed-szeged/cppbackport`.

The Author's Contributions

The author performed the literature review in the field of code transforming. He took part in defining the possible transformation scenarios, as well as taking part in their evaluation. The author designed and implemented the incremental framework. He designed the database scheme. He took part in implementing the transformations. The author also took part in the design and implementation of the test suite. He designed, implemented, and tested the methodology on how to use the framework as a pre-build step in developers' environment.

- **Gábor Antal**, Dávid Havas, István Siket, Árpád Beszédes, Rudolf Ferenc, and József Mihalicza. Transforming C++11 Code to C++03 to Support Legacy Compilation Environments In Proceedings of the IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM 2016), Raleigh, NC, USA. Pages 177–186, IEEE, October, 2016.

II. A Comparative Study on Static JavaScript Call Graph Algorithms

The contributions of this thesis point are discussed in Chapter 3.

The popularity and wide adoption of JavaScript both at the client and server side makes its code analysis more important than ever before. Most of the algorithms for vulnerability analysis, coding issue detection, or type inference rely on the call graph representation of the underlying program. Despite some obvious advantages of dynamic analysis, static algorithms should also be considered for call graph construction, as they do not require extensive test beds for programs; or their costly execution and tracing.

We systematically compared five widely adopted static algorithms – implemented by the npm call graph, IBM WALA, Google Closure Compiler, Approximate Call Graph (ACG), and Type Analyzer for JavaScript tools (TAJS) – for building JavaScript call graphs on 26 WebKit SunSpider benchmark programs and on 6 real-world Node.js modules. We provided a performance analysis as well as a quantitative and qualitative evaluation of the results.

Our findings include that there was a relatively large intersection of the found call edges among the algorithms, which proved to be 100% precise. However, most of the tools found edges that were missed by all others. ACG had the highest precision followed immediately by TAJS, but ACG found significantly more call edges. As for the combination of tools, ACG and TAJS together covered 99% of the found true edges by all algorithms, while maintaining a precision as high as 98%. Unfortunately, only two of the tools were able to analyze up-to-date multi-file Node.js modules due to incomplete language feature support. They agreed on almost 60% of the call edges, but each of them found valid edges that the other missed.

The Author's Contributions

The author did the research work in order to find the candidate tools and algorithms. He participated in designing the methodology. The author modified the tools in order to extract call graphs. He was also the developer of the format converter tool. Selecting the Node.js modules, and creating artificial large examples to stress test the tools were also his work. He took part in evaluating the results, and in their manual validation. He devised the methodology for the performance measurement, as well as conducting the performance analysis.

- **Gábor Antal**, Péter Hegedűs, Zoltán Tóth, Rudolf Ferenc, and Tibor Gyimóthy. Static JavaScript Call Graphs: A Comparative Study. In Proceedings of the 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 177–186, IEEE, Sep. 2018
  - **Distinguished Research Paper Award**

III. Combining Static and Dynamic Code Analysis with Machine Learning to Detect Software Issues in JavaScript Programs

The contributions of this thesis point are discussed in Chapter 4.

Issue prediction aims at finding source code elements in a software system that are likely to contain defects. Being aware of the most error-prone parts of the program, one can efficiently allocate the limited amount of testing and code review resources.

In this thesis point, we proposed two prediction models using different datasets and different features to predict software issues. We investigated how the state-of-the-art machine learning techniques, including a popular deep learning algorithm, perform in predicting functions with defects, such as possible bugs or vulnerabilities in JavaScript functions.

We applied 8 machine learning algorithms to build prediction models using a new dataset that we constructed for this research. We used static source code metrics as predictors and an extensive grid-search algorithm to find the best performing models. We also examined the effect of various re-sampling strategies to handle the imbalanced nature of the dataset. The best performing algorithm was KNN, with an F-measure of 0.76. Moreover, deep learning, tree, and forest-based classifiers, and SVM were competitive, with F-measures over 0.70. Although the F-measures did not vary significantly with the re-sampling strategies, the distribution of precision and recall did change.

We also proposed a function level JavaScript bug prediction model based on static source code metrics with the addition of hybrid (static and dynamic) code analysis based metrics of the number of incoming and outgoing function calls (HNII and HNOI). Our motivation for this is that JavaScript is a highly dynamic scripting language for which static code analysis might be very imprecise (as we have already seen in the case of call graphs), therefore, using purely static source code features for a prediction task might not be enough. We created a hybrid call graph analysis framework that analyzes the source code with various static and dynamic tools. We extracted 824 buggy and 1943 non-buggy functions from the publicly available BugsJS dataset [68] for the ESLint JavaScript project. We can confirm the positive impact of hybrid code metrics on the prediction performance of the ML models. Depending on the ML algorithm, applied hyper-parameters, and target measure we consider, hybrid invocation metrics bring a 2-10% increase in model performances (i.e., precision, recall, F-measure). Interestingly, replacing static NOI and NII metrics with their hybrid counterparts HNOI and HNII in itself improve model performances, however, using them all together yields the best results.

The created vulnerability dataset is available online: `https://inf.u-szeged.hu/~ferenc/papers/JSVulnerabilityDataSet`, while the framework is available on GitHub: `https://github.com/sed-szeged/hcg-js-framework`.

The Author's Contributions

The author participated in designing the methodology of this study. The literature review of the field was also done by the author. He took a major part in implementing the data collecting and merging tools. He also took part in the manual evaluation process. The design and the implementation of the hybrid call graph analysis framework were mainly the author's work. Creating the different feature sets was done by the author, He took part in the evaluation of the machine learning models' results.

- Rudolf Ferenc, Péter Hegedűs, Péter Gyimesi, **Gábor Antal**, Dénes Bán, and Tibor Gyimóthy. Challenging machine learning algorithms in predicting vulnerable JavaScript functions. In Proceedings of the 2019 IEEE/ACM

7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2019), pages. 8-14, IEEE, May 28, 2019

- **Gábor Antal**, Zoltán Tóth, Péter Hegedűs and Rudolf Ferenc. Enhanced Bug Prediction in JavaScript Programs with Hybrid Call-Graph Based Invocation Metrics. In Technologies 9, no. 1: 3, MDPI.

IV. Studying Typical Security Issues and Their Mitigation in Open-Source Projects

The contributions of this thesis point are discussed in Chapter 5.

Software security is undoubtedly a major concern in today's software engineering. Although the level of awareness of security issues is often high, practical experiences show that neither preventive actions nor reactions to possible issues are always addressed properly in reality. By analyzing large quantities of commits in the open-source communities, we can categorize the vulnerabilities mitigated by the developers and study their distribution, resolution time, etc. to learn and improve security management processes and practices. Moreover, understanding the typical vulnerabilities in programming languages helps researchers fine tune their machine learning models in predicting vulnerable software components.

In this thesis point, we used two different databases to mine vulnerability data. With the help of the Software Heritage Graph Dataset, we investigated the commits of two of the most popular script languages, Python and JavaScript. We distinguished the types of vulnerabilities (in terms of CWE groups) referred to in commit messages and compared their numbers within the two communities. We examined the average time elapsing between the publish date of a vulnerability and the first reference to it in a commit. We found that there is a large intersection in the vulnerability types mitigated by the two communities, but the most prevalent vulnerabilities are specific to language. Moreover, neither the JavaScript nor the Python community reacts very fast to appearing security vulnerabilities in general with only a couple of exceptions for certain CWE groups.

We also created several tools to help mine data needed for this and similar studies. We used these tools and showcased their capability of collecting data; we mined the most popular GitHub repositories and created our own database, which is publicly available. Our goal was to find out if there are common patterns within the most widely used programming languages in terms of security issues and fixes. We found that the same security issues might appear differently in different languages, and as such the provided solutions may vary just as much. We also found that projects with similar sizes can produce extremely different results, and have different common weaknesses, even if they provide a solution to the same task. These statistics may not be entirely indicative of the projects' standards when it comes to security, but they provide a good reference point of what one should expect.

The created tools and the dataset is available on GitHub: `https://cveminer.github.io`.

The Author's Contributions

The author devised the basic concepts of the study. He created and implemented the approach of mining data from the Software Heritage Graph Dataset and merging the results with the CVE/CWE data. Moreover, he laid the foundations for the implementation of the published, open-source tools. He also lead the further development of the tools. Merging and evaluating the results were done by the author. He took part in the manual validation of the results.

- ♦ **Gábor Antal**, Márton Keleti, and Péter Hegedűs. Exploring the Security Awareness of the Python and JavaScript Open Source Communities. In Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20). Association for Computing Machinery (ACM), New York, NY, USA, 16–20.
- ♦ **Gábor Antal**, Balázs Mosolygó, Norbert Vándor and Péter Hegedűs A Data-Mining Based Study of Security Vulnerability Types and Their Mitigation in Different Languages. In Proceedings of the International Conference on Computational Science and Its Applications (ICCSA 2020), Published in Lecture Notes in Computer Science (LNCS), vol 12252. Springer, Cham, page 1019-1034, Cagliari, Italy, July 1-4, 2020.

Table A.1 summarizes the main publications and how they relate to our thesis points.

№	[164]	[165]	[169]	[168]	[166]	[167]
I.	♦					
II.		♦				
III.			♦	♦		
IV.					♦	♦

Table A.1. Thesis contributions and supporting publications

B

Magyar nyelvű összefoglaló

Az új típusú koronavírus kitörését követően a kibertámadások száma naponta 300%-kal növekedett meg, így több, mint 4000 támadást követnek el minden nap [72]. A kiberbűnözők könyörtelenül célbavesznek mindenkit, beleértve a kórházakat, kutatókat, és az otthonról dolgozó embereket is [82, 87].

Ebben a disszertációban olyan témakörökre összpontosítunk, amelyek a szoftverminőség és szoftverbiztonság fontosságát hangsúlyozzák ki. Legelőször bemutatunk egy megoldást, amely segítségével több C++11 szabvány-beli elemet támogathatunk, szabványos C++03 környezetben. Ezt követően bemutatjuk összehasonlító vizsgálatunkat a statikus JavaScript hívási gráfok témakörében, amely bemutatja a népszerűbb hívási gráf építő algoritmusok közötti hasonlóságokat és különbségeket. Megmutattuk, hogy egy olyan rendkívül dinamikus nyelv esetében, mint a JavaScript, jelentős különbségek lehetnek az elkészült hívási gráfokban. Ezután bemutatjuk eredményeinket JavaScript-függvények biztonsági hibáinak előrejelzésében, amelyhez 8 jól ismert gépi tanulási algoritmust hasonlítottunk össze. Első lépésként csak statikus forráskód-metrikákat használtunk jellemzőként. Mivel biztató eredményeket kaptunk, kibővítettük a kutatásunk témakörét: bővítettük a modellünket hibrid (statikus és dinamikus) elemzésen alapuló hívási gráfból származó hívási metrikákkal. Végezetül pedig bemutatjuk a nyílt forrású rendszereken végzett tipikus sérülékenységekről és azok javításáról szóló tanulmányunkat. A témában összegyűjtött értékes információk segíthetnek a kutatóknak a gépi tanulási modellek finomhangolásában, a fejlesztőknek pedig abban, hogy jobban megismerjék a kódjukban lévő lehetséges sebezhetőségeket.

Az eredményeket négy tézispontba csoportosítottuk. A tézispontokhoz tartozó publikációkat a B.1. táblázat foglalja össze.

I. C++11 kód átalakítása C++03-ra örökölt fordítási környezetek támogatása érdekében

A tézisponthoz tartozó eredményeket a 2. fejezet tárgyalja.

Az újabb technológiák (pl.: programozási nyelvek, környezetek, függvénykönyvtárak) gyorsan változnak. Azonban különböző belső és külső megszorítások gyakran megakadályozzák, hogy a projektek (és fejlesztői csapatok) alkalmazkodjanak

ezekhez a változásokhoz. Ha naprakészek maradunk az újabb technológiákkal, akkor a szoftvereink kevésbé lesznek hibaérzékenyek, míg a teljesítményük javulni fog, mivel minden egyes frissítés számos új szolgáltatást nyújthat. Ennek ellenére előfordulhat, hogy az ügyfelek számára fontos az, hogy bizonyos platformokkal kompatibilis legyen az elkészült szoftver.

Jelen tézispont ezzel a problémával foglalkozik, a C++ programozási nyelv vonatkozásában. A Szegedi Tudományegyetem Szoftverfejlesztés Tanszékének egyik ipari partnere olyan fejlesztőkészleteket köteles használni, amelyek csak a C++ nyelv egy korábbi kiadását támogatják. A cég azonban szeretné lehetővé tenni a fejlesztők számára, hogy újabb nyelvi elemeket is használhassanak a kódjukban, és természetesen a fejlesztők is szívesen használnák a C++ újabb szabványait.

A probléma megoldásához először is alapos kutatást végeztünk, és összehasonlítottuk a jelen esetben használható lehetséges megoldásokat. Megterveztünk és megvalósítottunk egy olyan forráskód-transzformációs keretrendszert az LLVM és clang infrastruktúrán alapulva, amely képes a C++11 szabvány szerint írt forráskódot automatikusan átalakítani egy, az eredetivel funkcionálisan egyenértékű C++03 szabvány szerinti kódra. A keretrendszerünkkel a fejlesztőknek kihasználhatják az újabb nyelvi szabvány jelentős részét, miközben a termék éles (keretrendszerünkkel átalakított) változatának forráskódja továbbra is fordítható szabványos C++03 fordítóval. Az eszközt úgy terveztük meg, hogy egy programfutás során több transzformációt is képes legyen végrehajtani; támogatja az inkrementális futtatást (mindig csak az előző transzformáció óta módosult fájlok kerülnek átalakításra), a transzformációkat meghatározott sorrendben futtatjuk, így több transzformáció is képes párhuzamos futni. A keretrendszerünkben megvalósított transzformációk: Osztálydeklarációban történő adattag inicializálás; Fordításidejű típus következtetés (auto kulcsszó); Lambda függvények; Attribútumok; Final és override kulcsszavak; Intervallum-alapú számláló ciklus (for); Konstruktor delegálás; Típus elnevezés; *Kísérleti jelleggel megvalósított transzformációk.* A keretrendszerünket 4 nyílt és 2 zárt forráskódú ipari rendszeren is kiértékeltük. Bemutattuk az eszközünk teljesítményét különböző párhuzamosítási beállításokkal.

A megvalósításunk nyílt forráskódú és szabadon elérhető a GitHubon: `https://github.com/sed-szeged/cppbackport`.

A szerző hozzájárulása

A disszertáció szerzője végezte el a kódtranszformációval kapcsolatos szakirodalom feldolgozását. A szerző részt vett a lehetséges transzformációs forgatókönyvek meghatározásában, valamint azok kiértékelésében. Az inkrementalitást és transzformációfuttatást biztosító keretrendszert is a szerző készítette. Az adatbázissémát is ő tervezte meg. A transzformációs algoritmusok megvalósításában is aktívan részt vett. A szerző részt vett a teszteléshez szükséges keretrendszer és az általa futtatott tesztek megtervezésében és megvalósításában. Megtervezte, megvalósította és tesztelte azt a módszertant, amellyel a keretrendszer pre-build lépésként integrálható a fejlesztési folyamatokba.

♦ **Gábor Antal**, Dávid Havas, István Siket, Árpád Beszédes, Rudolf Ferenc, and József Mihalicza. Transforming C++11 Code to C++03 to Support

Legacy Compilation Environments In Proceedings of the IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM 2016), Raleigh, NC, USA. Pages 177–186, IEEE, October, 2016.

II. Statikus JavaScript hívási gráf építő algoritmusok összehasonlítása

A tézisponthoz tartozó eredményeket a 3. fejezet tárgyalja.

A JavaScript népszerűsége és széles körű elterjedtsége a kliens- és a szerveroldalon, minden eddiginél fontosabbá teszi az elemzését. A forráskódból kinyerhető hívási gráf számos sérülékenységvizsgáló programnak, kódolási szabálysértés ellenörzőnek, valamint típuskövetkeztető algoritmusnak az alapja. A dinamikus elemzés nyújtotta előnyök ellenére a statikus elemzés is megfontolandó, ha hívási gráf építésről van szó, ugyanis a statikus hívási gráf készítéséhez nem szükséges a program idő- és erőforrásigényes futtatása.

Összehasonlítottunk öt széles körben elterjedt statikus hívási gráf készítő algoritmust, amelyeket az npm call graph, IBM WALA, Google Closure Compiler, Approximate Call Graph (ACG), és a Type Analyzer for JavaScript (TAJS) eszközök valósítanak meg. Az összehasonlítást az eszközök által 26 WebKit SunSpider benchmark programon és 6 valós, nyílt forrású ipari Node.js rendszeren végeztük el. Elemeztük az eszközök teljesítményét, valamint elvégeztük az eredmények mennyiségi és minőségi kiértékelését.

Megállapítottuk, hogy az algoritmusok által megtalált hívási élek között viszonylag nagy átfedés van, ezek 100%-ban valós élek. A legtöbb eszköz azonban olyan éleket is talált, amelyet semelyik másik eszköz sem. Az ACG volt a legpontosabb, amelyet a TAJS követett, de az ACG lényegesen több hívási élt talált. Ami az eszközök kombinációját illeti, az ACG és a TAJS együttesen az összes algoritmus által talált valódi hívási élek 99%-át találta meg, 98%-os pontosság mellett. Sajnos csak két eszköz volt képes elemezni a több fájlból álló, modern Node.js projekteket, mivel a többi eszköz nem támogatta az újabb nyelvi elemeket. A hívási élek 60%-ban megegyeztek a két program által készített hívási gráfokban, de mindkét eszköz talált olyan valós hívási éleket, melyeket a másik eszköz nem.

A szerző hozzájárulása

A szerző kutatta fel a lehetséges hívási gráf készítő eszközöket. A szerző aktívan részt vett a kutatás módszertanának megtervezésében. Az eszközök módosítását is a szerző végezte el a hívási gráfok kinyerésének érdekében. Ő készítette el az eszközök kimenetét egységes formátumra konvertáló eszközt is. A Node.js modulok kiválasztása és az eszközök stressztesztteléséhez szükséges mesterségesen létrehozott nagy példák elkészítése is a szerző munkája volt. A disszertáció szerzője részt vett az eredmények kiértékelésében is, valamint azok manuális validációjában is. A szerző dolgozta ki a performanciamérés módszertanát, valamint annak elvégzése is a szerző saját munkája.

- **Gábor Antal**, Péter Hegedűs, Zoltán Tóth, Rudolf Ferenc, and Tibor Gyimóthy Static JavaScript Call Graphs: A Comparative Study. In Proceedings of the 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 177–186, Sep. 2018
  - **Distinguished Research Paper Award**

III. Statikus és dinamikus kódelemzés ötvözése gépi tanulással a JavaScript programok hibáinak felderítésére

A tézisponthoz tartozó eredményeket a 4. fejezet tárgyalja.

A hiba-előrejelző rendszerek célja, hogy megtalálják a szoftverrendszer azon részeit, amelyek valószínűleg hibát tartalmaznak. A program leginkább hibaérzékeny részeinek ismeretében hatékonyabban eloszthatjuk a korlátozott mennyiségben rendelkezésre álló tesztelési és kód review erőforrásokat.

Ebben a tézispontban két előrejelzési modellt készítettünk a szoftverproblémák előrejelzésére, melyek különböző adathalmazokon és különböző jellemzőkkel dolgoznak. Megvizsgáltuk, hogy a legkorszerűbb gépi tanulási technikák (köztük egy népszerű mélytanuló algoritmus) hogyan teljesítenek JavaScript függvények lehetséges sérülékenységeinek, valamint hibáinak előrejelzésében.

8 gépi tanulási algoritmust használtunk hiba előrejelzési modellek készítésére egy új, jelen kutatáshoz készített adathalmaz felhasználásával. Statikus forráskód metrikákat használtunk prediktorként és mindenre kiterjedő grid-search algoritmust használtunk a legjobban teljesítő modellek megtalálásának céljából. Az adathalmaz kiegyensúlyozatlanságának kezelésére megvizsgáltuk a különböző újramintavételezési (re-sampling) stratégiák hatását is. A legjobban teljesítő algoritmus a KNN volt, 0.76-os F-mértékkel (F-measure). Továbbá, a mélytanulás, a döntési fa, az erdő-alapú osztályozók, és az SVM is versenyképes volt, 0.70 feletti F-mértékkel. Annak ellenére, hogy az F-mértékek nem változtak jelentősen az újramintavételezési stratégiák változtatásával, a pontosság és fedés értékek változtak.

Továbbá megvalósítottunk egy függvény-szintű hiba-előrejelző modellt, amely JavaScript függvényekben található általános szoftverhibákat képes előrejelezni; ehhez a statikus forráskód metrikákon kívül bővítettük a jellemzők halmazát hibrid (statikus és dinamikus) kódelemzésen alapuló hívási metrikákkal (HNII (hibrid bejövő függvényhívások száma), HNOI (hibrid kimenő függvényhívások száma)). Mivel a JavaScript rendkívül dinamikus szkriptnyelv, így a statikus kódelemzési módszerek pontatlanok lehetnek. Ebből kifolyólag a pusztán statikus forráskódjellemzők használata a hiba-előrejelzési feladatokra nem biztos, hogy elegendő. Létrehoztunk egy hibrid hívásigráf-elemző keretrendszert, amely a forráskódot különböző statikus és dinamikus eszközökkel elemzi. Tanulmányunk alapján, amelyben az ESLint projekt 824 hibás és 1943 nem hibás függvényét választottuk ki a nyilvánosan elérhető BugsJS adathalmazból [68], megerősíthetjük a hibrid kódmetrikák pozitív hatását a modellek előrejelzési teljesítményére. A tanulóalgoritmustól, az alkalmazott hiperparaméterektől, valamint a vizsgált célmetrikától függően a hibrid hívási metrikák 2-10% javulást hoznak a modellek teljesítményében. Érdekes módon, a statikus NOI és NII metrikák helyettesítése hibrid megfelelőikkel már önmagában javítja a modellek teljesítményét, azonban a legjobb eredményt az összes metrika együttes használata adja.

A létrehozott sérülékenység adathalmaz nyilvánosan elérhető: `http://www.inf.u-szeged.hu/~ferenc/papers/JSVulnerabilityDataSet/`, míg a megvalósított keretrendszer a GitHubon található:
`https://github.com/sed-szeged/hcg-js-framework`.

A szerző hozzájárulása

A szerző részt vett a tanulmány módszertanának kialakításában. A szakterület irodalmának feldolgozása is a szerző munkája. Jelentős részt vállalt az adatgyűjtő és összevonó rendszer megtervezésében és kialakításában. A szerző részt vett a manuális kiértékelésben is. A hibrid hívási gráf elemzési keretrendszer megtervezése és megvalósítása főként a szerző munkája volt. A különböző jellemzőhalmazok kialakítása a szerző munkája. A gépi tanulási modellek eredményeinek kiértékelésében a szerző aktívan részt vett.

- Rudolf Ferenc, Péter Hegedűs, Péter Gyimesi, **Gábor Antal**, Dénes Bán, and Tibor Gyimóthy Challenging machine learning algorithms in predicting vulnerable JavaScript functions. In Proceedings of the 2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2019), pages. 8-14, IEEE, May 28, 2019
- **Gábor Antal**, Zoltán Tóth, Péter Hegedűs and Rudolf Ferenc. Enhanced Bug Prediction in JavaScript Programs with Hybrid Call-Graph Based Invocation Metrics. In Technologies 9, no. 1: 3, MDPI.

IV. Tipikus biztonsági hibák és javításuk vizsgálata nyílt forrású rendszerekben

A tézisponthoz tartozó eredményeket az 5. fejezet tárgyalja.

A szoftverbiztonság kétségtelenül napjaink egyik legfontosabb szempontja a szoftverfejlesztés területén. Annak ellenére, hogy a biztonsági sérülékenységekre gyakran nagyon ügyelnek a fejlesztők, a gyakorlat azt mutatja, hogy egy lehetséges hibára sem a megelőző, sem pedig a reakciós lépéseket sem feltétlenül hajtják végre helyesen. A nyílt forrású fejlesztőközösségek nagy mennyiségű commitjának elemzésével kategorizálhatjuk a fejlesztők által javított sebezhetőségeket, és tanulmányozhatjuk azok eloszlását, javítási idejét, és további jellemzőket a biztonságkezelési folyamatok és gyakorlatok megismerése és javítása érdekében. Ezenkívül a programozási nyelvek jellemző sérülékenységeinek megértése segítheti a kutatókat a sérülékenységeket előrejelző gépi tanulási modelleik finomhangolásában. Ebben a tézispontban két különböző adatbázist használtunk a sérülékenységi adatok bányászatához.

A Software Heritage Graph Dataset segítségével megvizsgáltuk két népszerű szkriptnyelv, a Python és a JavaScript projektjeinek commitjait. Megvizsgáltuk a commit-üzenetben említett sérülékenységek típusait (a CWE csoportok szerint) és összehasonlítottuk ezek számát a két közösségen belül. Megvizsgáltuk, az átlagosan eltelt időt egy sérülékenység publikálása és a commitokban való első hivatkozása között. Megállapítottuk, hogy a két közösség által javított sérülékenységtípusok között nagy átfedés van, de a leggyakoribb sebezhetőségtípusok jellemzőek egy adott nyelvre. Általánosságban elmondható, hogy egyik közösség nem reagál túl gyorsan a biztonsági problémákra. Azonban mindkét nyelv esetében találhatunk kivételeket, vagyis olyan CWE csoportokat, amikre a közösség gyorsabban reagál, mint más típusú sérülékenységekre. Létrehoztunk több eszközt, amelyek segítik a jelen és az ehhez hasonló tanulmányokhoz szükséges adatok bányászatát. Az általunk fejlesztett eszközök képességeit is bemutattuk; adatbányászat segítségével a legnépszerűbb GitHub repository-k felhasználásával létrehoztuk a saját adatbázisunkat, amely publikusan elérhető. Célunk az volt, hogy kiderítsük, vannak-e közös minták a programozási nyelveken belül a sérülékenységek

és javításuk tekintetében. Megállapítottuk, hogy ugyanazok a sérülékenységek különböző nyelvekben másképp jelenhetnek meg, így azok megoldási módja is eltérhet. Azt is megállapítottuk, hogy hasonló méretű projektek rendkívül eltérő eredményeket produkálhatnak, és különbözhetnek a jellemző sérülékenységtípusok még akkor is, ha a szoftverek ugyanarra a problémára nyújtanak megoldást. Habár ezek a statisztikák nem feltétlenül tükrözik teljes mértékben a projektek színvonalát a biztonság tekintetében, de jó kiindulási pontot nyújthatnak ahhoz, hogy mire lehet számítani egy adott projekt esetében.

A létrehozott eszközök és az adathalmaz elérhető a GitHubon: `https://cveminer.github.io`.

A szerző hozzájárulása

A disszertáció szerzője dolgozta ki a tanulmány alapvető koncepcióját. A Software Heritage Graph Dataset adatainak bányászata és az eredmények CVE/CWE adatokkal való összevonása is a szerző saját munkája. Ezenkívül ő fektette le a nyílt forrású adatbányászati eszközök megvalósításának alapjait. Az eszközök továbbfejlesztéséért is a szerző volt felelős. Az eredmények kiértékelése is főként a szerző munkája. Az eredmények kézi validálásában is aktívan részt vett a disszertáció szerzője.

- **Gábor Antal**, Márton Keleti, and Péter Hegedűs. Exploring the Security Awareness of the Python and JavaScript Open Source Communities. In Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20). Association for Computing Machinery (ACM), New York, NY, USA, 16–20.
- **Gábor Antal**, Balázs Mosolygó, Norbert Vándor and Péter Hegedűs A Data-Mining Based Study of Security Vulnerability Types and Their Mitigation in Different Languages. In Proceedings of the International Conference on Computational Science and Its Applications (ICCSA 2020), Published in Lecture Notes in Computer Science (LNCS), vol 12252. Springer, Cham, page 1019-1034, Cagliari, Italy, July 1-4, 2020.

№	[164]	[165]	[169]	[168]	[166]	[167]
I.	♦					
II.		♦				
III.			♦	♦		
IV.					♦	♦

B.1. táblázat. A tézispontokhoz kapcsolódó publikációk

Acknowledgement

Large part of the results of this dissertation were obtained in the SETIT Project (2018-1.2.1-NKP-2018-00004)[1].

The research was supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (MILAB).

[1]Project no. 2018-1.2.1-NKP-2018-00004 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.

Bibliography

[1] The callgraphjs tool. https://github.com/asgerf/callgraphjs.dart. Accessed: 2018-10-16.

[2] The code2flow tool. https://github.com/scottrogowski/code2flow. Accessed: 2018-10-16.

[3] escomplex - GitHub. https://github.com/escomplex/escomplex. Accessed: 2018-10-16.

[4] Espruino Website. https://www.espruino.com/. Accessed: 2018-10-16.

[5] Facebook Flow tool. https://github.com/facebook/flow. Accessed: 2018-10-16.

[6] GitHub octoverse website. https://octoverse.github.com. Accessed: 2018-10-16.

[7] Node Security Platform - GitHub. https://github.com/nodesecurity/nsp. Accessed: 2018-10-16.

[8] OpenStaticAnalyzer - GitHub. https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer. Accessed: 2021-04-29.

[9] Sunspider 1.0.2 benchmark. https://github.com/WebKit/webkit/tree/master/PerformanceTests/SunSpider/tests/sunspider-1.0.2. Accessed: 2018-10-16.

[10] The JavaScript Explorer Callgraph tool. https://github.com/shrivastava-apurva/Javascript-Explorer---Callgraph. [Online; accessed 04-February-2020].

[11] The v8 javascript engine website. https://v8.dev. [Online; accessed 29-April-2021].

[12] Vulnerability DB | Snyk. https://snyk.io/vuln. Accessed: 2018-10-16.

[13] Partial list of publications that rely on the WALA. http://wala.sourceforge.net/wiki/index.php/Publications, 2018. Accessed: 2018-10-16.

[14] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[15] Ibrahim Abunadi and Mamdouh Alenezi. Towards cross project vulnerability prediction in open source web applications. In *Proceedings of the The International Conference on Engineering MIS 2015*, ICEMIS '15, New York, NY, USA, 2015. Association for Computing Machinery.

[16] Gerald Aigner and Urs Hölzle. *ECOOP '96 — Object-Oriented Programming: 10th European Conference Linz, Austria, July 8–12, 1996 Proceedings*, chapter Eliminating virtual function calls in C++ programs, pages 142–166. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[17] Karim Ali and Ondřej Lhoták. Application-Only Call Graph Construction. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 688–712, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[18] Yasser Ali Alshehri, Katerina Goseva-Popstojanova, Dale G Dzielski, and Thomas Devine. Applying machine learning to predict software fault proneness using change metrics, static code metrics, and a combination of them. In *SoutheastCon 2018*, pages 1–7. IEEE, 2018.

[19] Ross Anderson, Chris Barton, Rainer Böhme, Richard Clayton, Michel J. G. van Eeten, Michael Levi, Tyler Moore, and Stefan Savage. *Measuring the Cost of Cybercrime*, pages 265–300. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[20] Gábor Antal, Zoltán Gábor Tóth, Péter Hegedűs, and Rudolf Ferenc. Enhanced Bug Prediction in JavaScript Programs with Hybrid Call-Graph Based Invocation Metrics (Training Dataset), November 2020.

[21] ASF+SDF meta-environment. `http://www.meta-environment.org/`. Accessed: 2016-06-21.

[22] Otto Skrove Bagge, Magne Haveraaen, and Eelco Visser. CodeBoost: A framework for the transformation of C++ programs. Technical report, 2001.

[23] Victor R Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[24] Gustavo E. A. P. A. Batista, Ronaldo C. Prati, and Maria Carolina Monard. A study of the behavior of several methods for balancing machine learning training data. *SIGKDD Explor. Newsl.*, 6(1):20–29, June 2004.

[25] Mihai Bazon. UglifyJS. 2016. Accessed: 2018-10-16.

[26] K. Bennett. Legacy systems: coping with success. *IEEE Software*, 12(1):19–23, Jan 1995.

[27] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[28] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based Analysis and Prediction for Software Evolution. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 419–429, June 2012.

[29] David Binkley, Henry Feild, Dawn Lawrie, and Maurizio Pighin. Increasing diversity: Natural language measures for software fault prediction. *Journal of Systems and Software*, 82(11):1793–1803, 2009.

[30] Matt Bishop. *Introduction to computer security*, volume 50. Addison-Wesley Boston, 2005.

[31] François Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In *In The second annual object-oriented numerics conference (OON-SKI*, pages 122–136, 1994.

[32] Michael Bolin. *Closure: The Definitive Guide: Google Tools to Add Power to Your JavaScript*. " O'Reilly Media, Inc.", 2010.

[33] Cagatay Catal, Akhan Akbulut, Sašo Karakatič, Miha Pavlinek, and Vili Podgorelec. Can we predict software vulnerability with deep neural network? In *Conference: Conference: 19th International Multiconference INFORMATION SOCIETY - IS 2016, At Ljubljana, Slovenia*, 10 2016.

[34] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.

[35] Kyriakos C. Chatzidimitriou, Michail D. Papamichail, Themistoklis Diamantopoulos, Michail Tsapanos, and Andreas L. Symeonidis. Npm-miner: An infrastructure for measuring the quality of the npm registry. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 42–45, New York, NY, USA, 2018. ACM.

[36] Ajay Chawla. Coronavirus (covid-19)–'zoom'application boon or bane. *Available at SSRN 3606716*, 2020.

[37] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui. Static detection of control-flow-related vulnerabilities using graph embedding. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 41–50, 2019.

[38] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[39] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[40] Clang Tools. `https://github.com/llvm-mirror/clang-tools-extra`. Accessed: 2016-04-17.

[41] Michael L Collard, Jonathan I Maletic, and Brian P Robinson. A lightweight transformational approach to support large scale adaptive changes. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[42] Common Vulnerabilities and Exposures. `https://cve.mitre.org/`, 2020. Accessed: 2020-02-04.

[43] Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In *iPRES 2017-14th International Conference on Digital Preservation*, pages 1–10, 2017.

[44] Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24, 2018.

[45] J Dijkstra. *Evaluation of Static JavaScript Call Graph Algorithms*. PhD thesis, Software Analysis and Transformation, 2014.

[46] Mikhail Dmitriev. Profiling Java Applications Using Code Hotswapping and Dynamic Call Graph Revelation. *SIGSOFT Softw. Eng. Notes*, 29(1):139–150, January 2004.

[47] DMS software reengineering toolkit by Semantic Designs. `https://www.semanticdesigns.com/Products/DMS/DMSToolkit.html`. Accessed: 2016-06-21.

[48] EDG C++ Front End. `https://www.edg.com/c`. Accessed: 2016-04-17.

[49] Frank Eichinger, Klemens Böhm, and Matthias Huber. Mining Edge-Weighted Call Graphs to Localise Software Bugs. In *Machine Learning and Knowledge Discovery in Databases*, pages 333–348, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[50] Frank Eichinger, Victor Pankratius, Philipp WL Große, and Klemens Böhm. Localizing Defects in Multithreaded Programs by Mining Dynamic Call Graphs. In *Testing–Practice and Research Techniques*, pages 56–71. Springer, 2010.

[51] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 602. IEEE Computer Society, 2001.

[52] Allen F. Interprocedural Data Flow Analysis. In *Information Processing 74 (Software)*, pages 398–402. North-Holland Publishing Co., Amsterdam, The Netherlands, 1974.

[53] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported Refactoring for JavaScript. *SIGPLAN Not.*, 46(10):119–138, October 2011.

[54] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 752–761, Piscataway, NJ, USA, 2013. IEEE Press.

[55] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 172–181. IEEE Computer Society, October 2002.

[56] Rudolf Ferenc, Tamás Viszkok, Tamás Aladics, Judit Jász, and Péter Hegedűs. Deep-water framework: The swiss army knife of humans working with machine learning models. *SoftwareX*, 12:100551, 2020.

[57] Javed Ferzund, Syed Nadeem Ahsan, and Franz Wotawa. Analysing bug prediction capabilities of static code metrics in open source software. In *Software Process and Product Measurement*, pages 331–343. Springer, 2008.

[58] Stephen Fink and Julian Dolby. WALA–The TJ Watson Libraries for Analysis, 2012.

[59] Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM Workshop on Large-Scale Attack Defense*, LSAD '06, page 131–138, New York, NY, USA, 2006. Association for Computing Machinery.

[60] Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey Shitov. Ultra lightweight javascript engine for internet of things. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 19–20, 2015.

[61] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. VulinOSS: a dataset of security vulnerabilities in open-source systems. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 18–21. ACM, 2018.

[62] GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`. Accessed: 2016-04-24.

[63] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[64] Rinkaj Goyal, Pravin Chandra, and Yogesh Singh. Impact of interaction in the combined metrics approach for fault prediction. *Software Quality Professional*, 15(3), 2013.

[65] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[66] David Gray, David Bowes, Neil Davey, Yi Sun, and Bruce Christianson. Using the support vector machine as a classification method for software defect prediction

with static code metrics. In *International Conference on Engineering Applications of Neural Networks*, pages 223–234. Springer, 2009.

[67] David Philip Harry Gray. Software defect prediction using static code metrics: formulating a methodology. 2013.

[68] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. BugsJS: a benchmark of javascript bugs. In *Proceedings of 12th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 90–101, 2019.

[69] Sonia Haiduc, Venera Arnaoudova, Andrian Marcus, and Giuliano Antoniol. The use of text retrieval and natural language processing in software engineering. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 898–899, 2016.

[70] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *2012 34th international conference on software engineering (ICSE)*, pages 200–210. IEEE, 2012.

[71] Include What You Use. `http://include-what-you-use.org/`. Accessed: 2016-04-17.

[72] Internet Crime Complaint Center (IC3) of US Federal Bureau of Investigation and United States of America. Internet Crime Report 2020. `https://www.ic3.gov`, 2020. Accessed: 2021-04-29.

[73] ISO/IEC 14882:2003 - Programming languages – C++. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=38110`. Accessed: 2016-04-23.

[74] ISO/IEC 14882:2011 - Information technology – Programming languages – C++. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372`. Accessed: 2016-04-23.

[75] Judit Jász, István Siket, Edit Pengő, Zoltán Ságodi, and Rudolf Ferenc. Systematic comparison of six open-source java call graph construction tools. 2019.

[76] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for Javascript. In *International Static Analysis Symposium*, pages 238–255. Springer, 2009.

[77] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[78] Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. Enabling the Continous Analysis of Security Vulnerabilities with VulData7. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 56–61, 2018.

[79] JSON Compilation Database Format Specification. `http://clang.llvm.org/docs/JSONCompilationDatabase.html`. Accessed: 2016-04-23.

[80] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, San Jose, CA, USA, July 2014. Tool demo.

[81] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 121–132, New York, NY, USA, 2014. ACM.

[82] Navid Ali Khan, Sarfraz Nawaz Brohi, and Noor Zaman. Ten deadly cyber security threats amid covid-19 pandemic, May 2020.

[83] Joris Kinable and Orestis Kostakis. Malware Classification Based on Call Graph Clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[84] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn, and Wolfgang E. Nagel. *Euro-Par 2011: Parallel Processing Workshops: CCPI, CGWS, HeteroPar, HiBB, HPCVirt, HPPC, HPSS, MDGS, ProPer, Resilience, UCHPC, VHPC, Bordeaux, France, August 29 – September 2, 2011, Revised Selected Papers, Part II*, chapter Scout: A Source-to-Source Transformator for SIMD-Optimizations, pages 137–145. Springer Berlin Heidelberg, 2012.

[85] D. Kuhn, Mohammad Raunak, and Raghu Kacker. An analysis of vulnerability trends, 2008-2016. pages 587–588, 07 2017.

[86] S. Kummita, G. Piskachev, J. Späth, and E. Bodden. Qualitative and quantitative analysis of callgraph algorithms for python. In *2021 International Conference on Code Quality (ICCQ)*, pages 1–15, 2021.

[87] Harjinder Singh Lallie, Lynsay A. Shepherd, Jason R.C. Nurse, Arnau Erola, Gregory Epiphaniou, Carsten Maple, and Xavier Bellekens. Cyber security in the age of covid-19: A timeline and analysis of cyber-crime and cyber-attacks during the pandemic. *Computers & Security*, 105:102248, 2021.

[88] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*, page 96. Citeseer, 2012.

[89] Jusuk Lee, Kyoochang Jeong, and Heejo Lee. Detecting metamorphic malwares using code graphs. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, page 1970–1977, New York, NY, USA, 2010. Association for Computing Machinery.

[90] Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. *Languages and Compilers for Parallel Computing (LCPC 2003), TX, USA, October 2-4, 2003. Revised Papers*, chapter Cetus – An Extensible Compiler Infrastructure for Source-to-Source Transformation, pages 539–553. Springer Berlin Heidelberg, 2004.

[91] Ond Lhoták et al. Comparing Call Graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 37–42. ACM, 2007.

[92] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2201–2215, New York, NY, USA, 2017. Association for Computing Machinery.

[93] Lianfa Li and Hareton Leung. Mining static code metrics for a robust prediction of software defect-proneness. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 207–214. IEEE, 2011.

[94] LLVM C Backend. `https://github.com/draperlaboratory/llvm-cbe`. Accessed: 2016-04-17.

[95] LLVM clang compiler infrastructure. `http://clang.llvm.org`. Accessed: 2016-04-17.

[96] Lech Madeyski and Marian Jureczko. Which process metrics can significantly improve defect prediction models? an empirical study. *Software Quality Journal*, 23(3):393–422, 2015.

[97] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical Static Analysis of Javascript Applications in the Presence of Frameworks and Libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 499–509. ACM, 2013.

[98] Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle. A Large-scale Cross-architecture Evaluation of Thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 11:1–11:11, New York, NY, USA, 2013. ACM.

[99] Ruchika Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504 – 518, 2015.

[100] Marangoni, Matthew, Wischgoll, and Thomas. Paper: Togpu: Automatic Source Transformation from C++ to CUDA using Clang/LLVM. *Electronic Imaging*, 2016(1):1–9, 2016-02-14T00:00:00.

[101] Carl Lawrence Mariano. *Benchmarking JavaScript Frameworks*. PhD thesis, Dublin Institute of Technology, 2017.

[102] Robert C. Martin. *The Clean Coder: A Code of Conduct for Professional Programmers*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2011.

[103] Fabio Massacci and Viet Hung Nguyen. Which is the right source for vulnerability studies? an empirical analysis on mozilla firefox. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, MetriSec '10, New York, NY, USA, 2010. Association for Computing Machinery.

[104] Wes McKinney et al. Pandas: a Foundational Python Library for Data Analysis and Statistics. *Python for High Performance and Scientific Computing*, 14(9), 2011.

[105] Nancy R Mead, Julia H Allen, Mark Ardis, Thomas B Hilburn, Andrew J Kornecki, Richard Linger, and James McDonald. Software assurance curriculum project volume 1: Master of software assurance reference curriculum. Technical report, CARNEGIE-MELLON UNIV. PITTSBURGH PA SOFTW. ENG. INST., 2010.

[106] P. Mell, K. Scarfone, and S. Romanosky. Common vulnerability scoring system. *IEEE Security Privacy*, 4(6):85–89, 2006.

[107] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[108] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, Inc., 1st edition, 2014.

[109] József Mihalicza. *Analysis and Methods for Supporting Generative Metaprogramming in Large Scale C++ Projects*. PhD thesis, Eötvös Loránd University, 2014.

[110] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient Javascript Mutation Testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 74–83, March 2013.

[111] MITRE Corporation. CWE - Common Weakness Enumeration. `https://cwe.mitre.org/`, 2020. [Online; accessed 29-April-2020].

[112] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie A. Williams. Challenges with applying vulnerability prediction models. In *HotSoS*, 2015.

[113] B. Mosolygó, N. Vándor, G. Antal, P. Hegedűs, and R. Ferenc. Towards a prototype based explainable javascript vulnerability prediction model. In *2021 International Conference on Code Quality (ICCQ)*, pages 15–25, 2021.

[114] M. Mossienko. Automated cobol to java recycling. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, pages 40–50, 2003.

[115] Nuthan Munaiah and Andrew Meneely. Beyond the attack surface: Assessing security risk with random walks on call graphs. In *SPRO '16*, 2016.

[116] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An Empirical Study of Static Call Graph Extractors. *ACM Trans. Softw. Eng. Methodol.*, 7(2):158–191, April 1998.

[117] Vincenzo Musco, Martin Monperrus, and Philippe Preux. A large scale study of call graph-based impact prediction using mutation testing. *Software Quality Journal*, 25, 07 2016.

[118] LLB Muthuppalaniappan, Menaka and Kerrie Stevenson. Healthcare cyber-attacks and the COVID-19 pandemic: an urgent threat to global health. *International Journal for Quality in Health Care*, 33(1), 09 2020. mzaa117.

[119] Jaechang Nam. Survey on software defect prediction. *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep*, 2014.

[120] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 529–540, 01 2007.

[121] Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, page 3. ACM, 2010.

[122] Modules | Node.js v13.7.0 Documentation. `https://nodejs.org/api/modules.html#modules_folders_as_modules`, 2020. Accessed: 2020-02-04.

[123] National Vulnerability Database. `https://nvd.nist.gov/`, 2020. Accessed: 2020-02-04.

[124] Juan C. Oliveros. VENNY. An interactive tool for comparing lists with Venn diagrams. http://bioinfogp.cnb.csic.es/tools/venny/index.html.

[125] Edit Pengő and Péter Gál. Grasping primitive enthusiasm - approaching primitive obsession in steps. In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT)*, pages 423–430, 2018.

[126] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. The software heritage graph dataset: Public software development under one roof. In *MSR 2019: The 16th International Conference on Mining Software Repositories*, pages 138–142. IEEE, 2019.

[127] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. The Software Heritage Graph Dataset: Large-scale analysis of public software development history. In *MSR 2020: The 17th International Conference on Mining Software Repositories*. IEEE, 2020.

[128] Mark Pilgrim and Simon Willison. *Dive Into Python 3*, volume 2. Springer, 2009.

[129] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[130] Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: Dynamic Type Inconsistency Analysis for Javascript. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 314–324, Piscataway, NJ, USA, 2015. IEEE Press.

[131] Sanjeev Kumar Punia, Anuj Kumar, and Amit Sharma. Evaluation the quality of software design by call graph based metrics. *Global Journal of Computer Science and Technology*, 2014.

[132] K Punitha and S Chitra. Software defect prediction using software metrics-a survey. In *2013 International Conference on Information Communication and Embedded Systems (ICICES)*, pages 555–558. IEEE, 2013.

[133] Python Package Index. `https://pypi.org/`, 2020. Accessed: 2020-02-04.

[134] Packaging and distributing projects - Python Packaging User Guide. `https://packaging.python.org/tutorials/packaging-projects`, 2020. Accessed: 2020-02-04.

[135] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and software technology*, 55(8):1397–1418, 2013.

[136] Abhijit Rao and Steven J Steiner. Debugging From a Call Graph, January 22 2013. US Patent 8,359,584.

[137] Mozilla Rhino. JavaScript for Java. `http://www.mozilla.org/rhino`, 2018. Accessed: 2018-10-16.

[138] ROSE compiler infrastructure. `http://rosecompiler.org/`. Accessed: 2016-06-21.

[139] B. G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, 1979.

[140] M. Shahzad, M. Z. Shafiq, and A. X. Liu. A Large Scale Exploratory Analysis of Software Vulnerability Life Cycles. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 771–781, June 2012.

[141] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Softw. Eng.*, 37(6):772–787, November 2011.

[142] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 315–317. ACM, 2008.

[143] Yonghee Shin and Laurie A. Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18:25–59, 2011.

[144] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2012.

[145] Miltiadis Siavvas, Dionisis Kehagias, and Dimitrios Tzovaras. A preliminary study on the relationship among software metrics and specific vulnerability types. In *2017 International Conference on Computational Science and Computational Intelligence – Symposium on Software Engineering (CSCI-ISSE)*, 12 2017.

[146] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, page 1–5, New York, NY, USA, 2005. Association for Computing Machinery.

[147] H. M. Sneed. Migrating from cobol to java. In *2010 IEEE International Conference on Software Maintenance*, pages 1–7, 2010.

[148] Stratego/XT program transformation language. `http://strategoxt.org/`. Accessed: 2016-06-21.

[149] Sudo vulnerability in macOS. `https://www.techradar.com/news/linux-and-macos-pcs-hit-by-serious-sudo-vulnerability`, 2020. [Online; accessed 04-February-2020].

[150] Brian J Sullivan. Method and apparatus for converting cobol to java, September 17 2002. US Patent 6,453,464.

[151] Installing Encore (Symfony Docs). `https://symfony.com/doc/current/frontend/encore/installation.html#installing-encore-in-non-symfony-applications`, 2020. [Online; accessed 04-February-2020].

[152] The TXL programming language. `http://www.txl.ca/`. Accessed: 2016-06-21.

[153] Mario Linares Vásquez, Gabriele Bavota, and Camilo Escobar-Velasquez. An empirical study on android-related vulnerabilities. *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 2–13, 2017.

[154] Ju An Wang and Minzhe Guo. Vulnerability categorization using bayesian networks. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, New York, NY, USA, 2010. Association for Computing Machinery.

[155] Shiyi Wei and Barbara G Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 336–346. ACM, 2013.

[156] Tim Weil and San Murugesan. It risk and resilience-cybersecurity response to covid-19. *IT Prof.*, 22(3):4–10, 2020.

[157] Frank Wilcoxon, SK Katti, and Roberta A Wilcox. Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test. *Selected tables in mathematical statistics*, 1:171–259, 1970.

[158] Tao Xie and David Notkin. An Empirical Study of Java Dynamic Call Graph Extractors. *University of Washington CSE Technical Report 02-12*, 3, 2002.

[159] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song. Spain: Security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 462–472, May 2017.

[160] Jeff Younker. *Foundations of agile Python development.* Apress, 2009.

[161] L. Yu. Empirical study of python call graph. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1274–1276, 2019.

[162] Zhe Yu, Christopher Theisen, Hyunwoo Sohn, Laurie Williams, and Tim Menzies. Cost-aware vulnerability prediction: the HARMLESS approach. *CoRR*, abs/1803.06545, 2018.

[163] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third International Conference onSoftware Testing, Verification and Validation (ICST)*, pages 421–428. IEEE, 2010.

Corresponding Publications of the Author

[164] G. Antal, D. Havas, I. Siket, Á. Beszédes, R. Ferenc, and J. Mihalicza. Transforming c++11 code to c++03 to support legacy compilation environments. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 177–186, 2016.

[165] G. Antal, P. Hegedus, Z. Tóth, R. Ferenc, and T. Gyimóthy. Static javascript call graphs: A comparative study. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 177–186, Sep. 2018.

[166] Gábor Antal, Márton Keleti, and Péter Hegedűs. Exploring the security awareness of the python and javascript open source communities. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 16–20, New York, NY, USA, 2020. Association for Computing Machinery.

[167] Gábor Antal, Balázs Mosolygó, Norbert Vándor, and Péter Hegedűs. A data-mining based study of security vulnerability types and their mitigation in different languages. In *International Conference on Computational Science and Its Applications*, pages 1019–1034. Springer, 2020.

[168] Gábor Antal, Zoltán Tóth, Péter Hegedűs, and Rudolf Ferenc. Enhanced bug prediction in javascript programs with hybrid call-graph based invocation metrics. *Technologies*, 9(1):3, 2021.

[169] R. Ferenc, P. Hegedűs, P. Gyimesi, G. Antal, D. Bán, and T. Gyimóthy. Challenging machine learning algorithms in predicting vulnerable javascript functions. In *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, pages 8–14, 2019.