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

Summary of the Ph.D. Thesis

University of Szeged
Ph.D. School in Computer Science

Introduction

Nowadays, practically everything is powered by software. Everyone uses them, whether for daily work or leisure. Because of this, the amount of software being built is increasing. After the outbreak of the new coronavirus, the number of daily cyber-attacks grew by 300%, causing more than 4,000 attacks a day [5]. Cyber-criminals are mercilessly targeting everyone. A great example is the Zoom bombing exploit [11]. The vulnerability allowed attackers to intercept authentication and join any conversation they wanted in the Zoom video calling software. This example shows that we must pay attention to software quality and safety. However, as we are human beings, we occasionally make mistakes in our code that might go unnoticed for a long time. Ideally, we have enough time and freedom to do our best in order to develop quality software. Unfortunately, this state hardly exists in real life.

The customer might 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.

In order to help developers achieve more freedom in using newer language standards, we studied this field and found that there are not any tools that help developers use a newer standard of C++ in such a way that the code becomes compatible with an older standard of the language automatically. We designed and implemented a complete solution to aid this problem. Our tool is able to convert the source code so that it complies with the C++03 language standard.

Of course, this is not the only problem developers have to face from time to time. With the increasing use of dynamically typed languages (such as JavaScript and Python), the analysis of them is more pressing than ever. 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. Many code analysis tools rely on the call graph representation of the program. The call graph is the basis for many other, more complex data structures (such as the control flow graph), which are essential in order to detect issues in a given program. Therefore, the precision of call graphs is extremely important. As 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, and presented our findings.

Using source code metrics for predicting software issues is quite a mature technique [9, 10, 3, 7]. 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 even on a more fine-grained level), developers could use it to prevent issues in their code. But the practical adoption of 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 presented a comprehensive comparison of 8 well-known machine learning algorithms on predicting vulnerable JavaScript functions. 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 function invocation metrics with their counterparts coming from static and dynamic analysis (i.e. hybrid analysis).

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. We used the Software Heritage Graph Dataset [8] to mine data, in order to help JavaScript and Python developers learn their languages' typical security issue types and their characteristics. We also defined an approach on how to mine data for this purpose. We provided a toolset, and we presented our findings on the typical security issues and their characteristics in several programming languages.

The thesis consists of four thesis points. In this booklet, we summarize the results of each thesis point.

№	[12]	[13]	[17]	[16]	[14]	[15]
I.	◆					
II.		◆				
III.			◆	◆		
IV.					◆	◆

Table 1: Thesis contributions and supporting publications

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

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, 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, using LLVM and clang infrastructure. 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 transformation framework 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. Based on this list, the transformation engine performs the necessary changes. We had to take into consideration that there are numerous 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. The basic operation of the framework is as follows:

- 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.
- It then iterates over the list of transformations.
- 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.

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*. An example of transforming a lambda function is shown in Figure 1.

```
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 1: Lambda function example

We evaluated our transformation framework from two aspects: correctness of the transformed code and performance (runtime). 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.

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.

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 popularity and wide adoption of JavaScript both at the client- and server-side makes its code analysis more important than ever before. 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. We can use call graphs as a basis for further analysis, for example, a full interprocedural control flow graph (ICFG) can be built upon the call graph. 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). 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. In this thesis point, 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 in order to have a deeper understanding about the state-of-the-art static call graph construction algorithms for JavaScript.

We had to modify some of the tools, mainly to extract and dump the call graphs built into the memory of the programs. Next, we collected the produced outputs of the tools and converted them into a unified, JSON-based format. We created a merged JSON with the same structure using our graph comparison tool. 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 statistics on these individual and merged JSON files. To perform a deep comparison of the tools, we identified three test input groups: real-world, single file examples (the SunSpider benchmark of the WebKit browser engine [2]); real-world, multi-file Node.js modules (6 real-world, widely used Node.js modules that use new language features); and generated large examples (they contain numerous functions and calls between them).

For the qualitative analysis – inspired by the work of Lhoták et. al [6] –, we created a call graph comparison script written in Python. Besides comparing the results, we evaluated all the 348 call edges found by the five tools on the 26 SunSpider benchmark programs. As for the Node.js modules, the large number of edges made it impossible to validate all of them. We selected a statistically significant representative random sample of edges to achieve a 95% confidence level with a 5% margin of error.

Out of 348 call edges found by any of the tools, 257 were true edges. 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. As we systematically evaluated all 348 found call edges, we could also calculate precision and recall values for each tool and their arbitrary combinations.

Table 2: Precision and recall measures for tools

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%

Table 2 contains the detailed statistics of the tools, while Table 3 contains the top combinations of the tools, ordered by their F-measure. The first column 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. The fifth (Prec.), sixth (Rec.*), and seventh (F) columns contain the precision (TP / All), recall (TP* / TP) and F-measure values, respectively.

From the individual tools, ACG stands out with its almost perfect precision and quite high recall values. While TAJS and npm-cg maintain similarly high precision, their recalls are far below ACG's. Closure's recall is very close to that of ACG, but it has significantly lower precision. WALA has moderate precision, but the worst recall 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). There are no other three-, four-, or five-tool combinations that would even come close to this F-measure score. Taking all the tools into consideration, the combined precision decreases to 74% with a perfect recall.

As we already described, only ACG and Closure were able to analyze the state-of-the-art Node.js modules. 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.

Table 3: Top precision and recall measures for the combinations of the tools

Tool(s)	**TP**	**All**	**TP***	**Prec.**	**Rec.***	**F**
ACG+TAJS	254	260	257	98%	99%	98%
npm-cg+ACG+TAJS	255	279	257	91%	99%	95%
ACG+WALA+TAJS	254	279	257	91%	99%	95%
ACG+WALA	241	262	257	92%	94%	93%
npm-cg+ACG	239	259	257	92%	93%	93%
npm-cg+WALA+TAJS	238	258	257	92%	93%	92%
npm-cg+ACG+WALA+TAJS	255	298	257	86%	99%	92%
npm-cg+TAJS	233	255	257	91%	91%	91%
ACG+Closure	255	309	257	83%	99%	90%
npm-cg+ACG+WALA	242	281	257	86%	94%	90%
ACG+Closure+TAJS	257	311	257	83%	100%	90%

Each tool had its strengths and weaknesses. Our purpose was not to declare a winner, rather to gain empirical insights into the capabilities and effectiveness of the state-of-the-art static call graph extractors.

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

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, both vulnerability and bug prediction can support software maintenance and evolution to a great extent.

In this thesis point, we proposed two prediction models using different datasets and different features to predict software issues. We investigated whether or not predicting defects in functions is feasible based on various software metrics. We compared the performances of the most widely

used machine learning algorithms on this prediction task, including two deep neural network variants (DNN_s, DNN_c), 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 applied various re-sampling strategies to handle the imbalanced nature of the dataset.

First, we investigated how the machine learning techniques perform in predicting functions with possible security vulnerabilities in JavaScript programs. To the best of our knowledge, there were no existing vulnerability datasets for JavaScript programs specifically, so we created a fine-grained, public JavaScript vulnerability dataset with data extracted from several vulnerability databases 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.

Table 4: 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

We used static source code metrics as predictors and an extensive grid-search algorithm to find the best performing models. 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 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. A simple baseline algorithm (which predicts all instances to be vulnerable) achieved a precision of 0.12 and a perfect recall of 1, which adds up to an F-measure of 0.21. The results are summarized in Table 4. 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.

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 extracted 824 buggy and 1,943 non-buggy functions from the publicly available BugsJS dataset [4] for the ESLint JavaScript project. We created a hybrid call graph analysis framework that uses the source code

of the project as input. Then we analyzed the source code with various static and dynamic tools (which might require running the source code itself, and processing the obtained execution logs). Following the analyses, the framework converts all the tool-specific outputs to a unified JSON format. We used the unified JSON to calculate hybrid invocation metrics (i.e., HNII and HNOI). Besides computing the hybrid metrics, a standard set of metrics is provided by a static source code analyzer named OpenStaticAnalyzer [1]. Based on our results, we can confirm the positive impact of hybrid code metrics on the prediction performance of the ML models.

Table 5: The best results of the nine ML models according to their F-measure

ML algorithm	**Feature set**	**Accuracy**	**Precision**	**Recall**	**F-measure**	**MCC**
Forest	S+H	0.816	0.753	0.569	0.648	0.54
KNN	S+H	0.788	0.646	0.635	0.641	0.49
DNN_c	S+H	0.784	0.649	0.601	0.624	0.47
Tree	S+H	0.781	0.649	0.58	0.612	0.46
DNN_s	H	0.774	0.634	0.569	0.6	0.44
Logistic	S+H	0.787	0.682	0.533	0.598	0.46
SVM	S+H	0.789	0.699	0.515	0.593	0.47
Linear	S+H	0.769	0.67	0.443	0.533	0.4
Bayes	S+H	0.772	0.713	0.394	0.508	0.4

Table 5 shows the best prediction performances (i.e., models with best performing hyper-parameters and feature set) of nine machine learning algorithms according to their F-measures. 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 publicly available online: `https://inf.u-szeged.hu/~ferenc/papers/JSVulnerabilityDataSet`, while the proposed framework can be found 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. He also took part in creating the different feature sets, as well as taking 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

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, and other characteristics to learn and improve security management processes and practices. Moreover, understanding the typical vulnerabilities in programming languages can help researchers fine-tune their machine learning models in predicting vulnerable software components. In this thesis point, we used two different databases to mine and analyze vulnerability data. We used the CVE catalog to identify vulnerabilities. CVEs (short for Common Vulnerabilities and Exposures) 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), which is a widely adopted categorization for vulnerabilities.

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 identified commits that mitigate a certain vulnerability in the code (i.e. vulnerability resolution commits). We examined how quickly the JavaScript and Python communities mitigate a newly published security vulnerability. 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 found that the JavaScript projects refer to security vulnerabilities falling into 87 different categories, the Python projects to 71, out of which 55 categories are common. Despite the large intersection in the security vulnerability types, the number of mitigated vulnerabilities differs 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 2).

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. 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 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.

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 (according to GHTorrent) for several programming languages and created our own

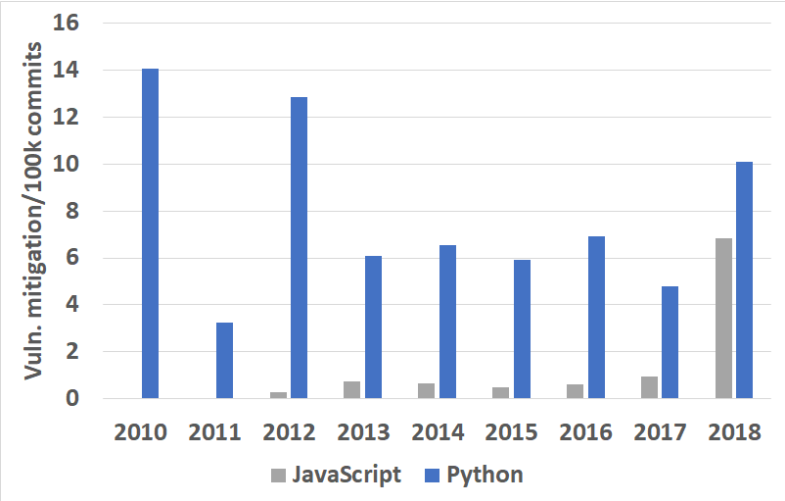

Figure 2: Vulnerability mitigation ratio per year

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.

Our findings include 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. 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. We also found that the correlation between the severity of CVEs and the time it took to fix them shows how prepared developers were when it came to fixing these vulnerabilities. As we experienced, there is no strong correlation between the severity and the vulnerability fixing time, however, smaller correlations exist. 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.

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 Interna-

tional 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.

Summary

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.

Acknowledgements

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.

A large part of the results of this dissertation was 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).

Gábor Antal, 2021

[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.

References

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

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

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

[4] 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.

[5] 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.

[6] 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.

[7] Jaechang Nam. Survey on software defect prediction. *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep*, 2014.

[8] 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.

[9] 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.

[10] 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.

[11] Tim Weil and San Murugesan. It risk and resilience-cybersecurity response to covid-19. *IT Prof.*, 22(3):4–10, 2020.

Corresponding Publications of the Author

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.