Evolution of Security in Automated Migration Processes

In memory of Mahsa Amini

and all those who were innocently killed ...

Evolution of Security in Automated Migration Processes

By Seyed Parsa Tayefeh Morsal, B.Sc

A Thesis Submitted to the School of Graduate Studies in the Partial Fulfillment of the Requirements for the Degree Master of Applied Science

McMaster University

Master of Applied Science (2022)

Hamilton, Ontario (Department of Computing & Software)

TITLE: Evolution of Security in Automated Migration Processes

AUTHOR: Seyed Parsa Tayefeh Morsal (McMaster University)

SUPERVISOR: Dr. Richard Paige

NUMBER OF PAGES: ix, 75

Abstract

As users' requirements change in today's fast-paced business market, computer software has to adapt to new hardware, technologies and requirements to keep up with the trend. Therefore, to avoid depreciation and obsolescence, which can have detrimental effects on a product, software needs to be constantly maintained and, when passed a certain point in its lifecycle, needs to be migrated or re-developed from scratch. Automated migration enables software vendors to decrease the cost of the migration process by source code generation. However, as security is a crucial requirement in any system, it is not guaranteed that the previously satisfied security requirements are satisfied in the migrated software. Therefore, it is critical to study the evolution of security throughout the automated migration process to predict where new security vulnerabilities may emerge and to understand the scale on which the security is affected.

Acknowledgements

I want to express my utmost gratitude to my primary supervisor, Prof. Richard Paige, who guided me throughout my academic journey with his deep insight, exceptional professionalism and unconditional support during the difficult times of COVID and immigration.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Security is a critical property of any system, and a considerable amount of resources are spent to guarantee and maintain a system's security. However, as the system changes due to obsolescence or change in users' requirements, the previously satisfied security requirements may no longer hold. Therefore, it is critical to understand how security is affected when a system goes through any change.

Migration is one of the most common and essential changes that can happen to a system throughout its lifetime [1]. A migration process can be applied on various levels, such as hardware migration in case of hardware obsolescence, operating system migration due to reliability or compatibility issues, database migration due to potential changes in business requirements, or programming language migration due to language or library obsolesce. Regardless of the scope of the migration process, it is

critically important to know whether functional and non-functional requirements within the scope of the requirements of the migrated system are still satisfied after the migration process is completed. The system validation and verification are mainly achieved and investigated through unit testing. However, this method only answers a single case: whether the functional and non-functional requirements are still satisfied in this particular system that went through a specific migration process [1].

Although this traditional method provides a case-dependant answer, it fails to generalize for future migrations or estimate how a specific functional or non-functional property is affected. Most importantly, in this method, in order to verify whether critical system properties such as security remain intact or were affected by the migration process, the migration process should be completed; as a result, if it fails to satisfy critical system requirements, it will waste the resources used to complete the migration process in the first place [2].

Therefore, a reasonable estimation of how much a system requirement, whether function or non-functional, is affected by a migration process before fully committing to the migration process serves a great advantage and may avoid potential resource loss due to a failed migration process.

In this thesis, it is the author's main goal to study how security is affected in an automated programming language migration by studying various cases of automated language migration and try to achieve a generalized answer on the extent to which security is affected in such process.

1.2 Research Questions

The main question that this thesis hopes to answer within its scope is how an automated migration process affects the security of a software product, among its other non-functional properties. The literature review carried out (see Chapter 2) suggests that the extent to which security properties are affected by software migration has not been systematically analyzed in previous work. Although the answer to this question can be case-dependent and have various dimensions, the goal of this thesis is to provide potential software developers who might incorporate an automated migration process to migrate their obsolete codebase to a new language with an estimate of what the state of their software product would be after the migration, from the security standpoint.

1.3 Motivations

Although the answer this thesis hopes to provide to the above research question would be generalized and can be, in some cases, inaccurate due to the different nature of every software product, the author believes that even a rough estimate of the level of security concern the software developers would face after migration can be immensely beneficial to them to make critical technical decisions before actually going through the effort-intensive process of complete production level migration. For instance, using the analytical framework introduced in this thesis, a team of software developers might decide that given the results of the post-migration security analysis,

and considering the internal efforts of the team to apply the migration process, as well as any other hidden factors, know only to the team members, it might not be beneficial for them to go through an automated migration process or move to a model-driven architecture. Even for a mid-size corporate, knowing the fact that whether or not they will need to re-engineer their product's security model after migration, or in contrast, whether the security concerns would be minimal and could be dealt with in a short period, can make a substantial financial difference for them, where the former may require hundreds of thousands of dollars worth of reinvestment. At the same time, the latter can be done with the company's ongoing payrolls.

1.3.1 Structure of the thesis

In Chapter 2, related background concepts are briefly presented. Chapter 3 provides a review of the related literature and previous works. The author's proposed method is introduced in Chapter 4, and technical details, limitations and challenges are discussed. Chapter 5 provides a detailed description of the performed experiments and samples, followed by the observations and results of the experiments in Chapter 6. Finally, in chapter 7 author's conclusion and final discussion are presented.

Chapter 2

Background

2.1 Software Obsolescence

Like any other product, software systems require constant maintenance and support to adapt to their users' ever-changing needs. However, as this is an effort-intensive process, software vendors might fall behind in keeping up with the innovation waves as customers' requirements progress. This can be caused by multiple factors such as a lack of financial resources to maintain the product, obsolescence of hardware systems used to develop the software product, or the technical team's inability to incorporate new technologies. All of which can gradually deteriorate the quality of a product that once perfectly served its customers' needs and result in the total loss of efforts made to develop and maintain the software product in the first place [3].

Therefore, software obsolescence is a serious problem that calls for extensive engineering and analysis; otherwise, if taken lightly, it can turn

revenue-making software into a constant liability for its vendors or, in worst-case scenarios, lead to disasters in case of software failure in safety-critical systems [4].

2.2 Mitigation Strategies

One of the most effective ways to mitigate software obsolescence is to try to avoid it in the first place through frequent system upgrades and code maintenance. This includes identifying deprecated APIs and libraries and incorporating modular architectures to reduce coupling and dependency between different modules. Decoupling the modules reduces the effort needed to replace or redesign a deprecated module. However, another source of software obsolescence is the technological gap between different parts of the product, which in the case of the highly modular system can result in sudden incompatibility between different modules once a technology used by a specific module is deprecated or the service is discontinued [5].

However, applying obsolescence mitigation strategies is naively overlooked since crippling obsolescence does not seem to be an imminent threat to a working system or service until the technological gap between different parts of the system or availability of required third-party hardware or service compromises the main functional properties of the product. By this time, an extensive system-wide migration is required to maintain the software system [5].

2.3 Migration

As mentioned earlier, migration is one of the main obsolescence mitigation strategies, which ranges from hardware migration, i.e. using new or more compatible hardware, migrating to a different operating system, migrating to a new programming language or using updated and frequently maintained APIs and libraries. Migration can also be a system-wide transition, such as moving from a monolithic architecture to a microservice architecture or incorporating a model-driven design [1]. However, as mentioned earlier, any of the migrations are effort-extensive and require complex changes in the system, the difficulty of which might outweigh the required effort to recreate the software system from scratch in the new desired platform or language. In some cases, if possible, rewriting the codebase from scratch in a modern language can be the optimal solution, especially when the technological and age gap between the source language and the target language is enormous, e.g. migrating from COBOL to Javascript. However, in other cases, this may not be a feasible solution, significantly when rewriting the entire codebase might take substantial time and resources that are unavailable or the obsolete part of the system is entirely or partially isolated from the rest of the system [1] [2].

Many mitigating solutions for addressing software obsolescence and updating a software system are accessible from a technological standpoint. For example, software rehosting and systematic migration solve the issue by moving a software system to a new platform or development environment

simultaneously and in a scheduled manner while taking software architecture into account [6].

2.4 Model-Driven Engineering

Model-driven engineering (MDE) is a software engineering approach that mainly focuses on raising the level of abstraction to make the general course of designing a software product easier. MDE introduces models into designing and maintaining a software system, which allows developers and stakeholders to deal with design decisions on a more abstract level, preventing the design process from becoming unnecessarily overcomplicated with technical and implantation details [7]. MDE is not limited to the early stages of a software system. By focusing on models instead of other software artifacts such as code and documentation, MDD introduces a new principle to the entire software evolution cycle, from designing an initial software product to maintaining a software system through its lifetime and migrating an existing software system when it faces obsolescence. By incorporating models not weighed down by technical and syntactic details, MDE sets the ground for the automation of various software processes [8]. For instance, model-to-model transformation enables automated modification of models, while model-to-text transformation allows the automated generation of code from the existing models [9].

2.5 Automated Software Migration

Automated software migration is migrating a source software into a new language, platform or environment without manually rewriting each line of code in the target language. Instead, in an automated migration process, most of the codebase is automatically converted to a new language using methods similar to parsing and compiling the source code into a target machine code [10]. However, automated migration is not limited to transpiling a source code in a source language to a code with similar functionality in the target language. If the source software has been developed using MDE principles and techniques, then redevelopment and migration time may be significantly reduced because of the opportunity to reuse model-to-model and model-to-text transformations. It must be noted that only in straightforward cases automated migration directly results in executable code in a target language. The output of the process usually cannot be compiled and can be, to varying extents, incomplete and still requires modification and completion to become executable. However, the overall effort to modify the automatically generated code is promised to be significantly lower than rewriting the whole software from scratch in the target language [11]. The automatically generated code promises to hold, if not all, most of the functional properties of the source software. The automated migration process allows the development team to exploit the potentials of transpilers and model-to-text transformers to do most of the effort-intensive job of converting code bodies into the target language.

2.6 Non-functional Requirements

Non-functional requirements (NFR) describe limitations on the solution space and encompass a broad spectrum of attributes like security, maintainability, usability, execution time, and safety. In contrast, functional requirements specify what the system needs to perform [12]. NFRs are system-wide properties that cannot be directly pinned down to a single component in the system; in contrast, they are satisfied when each component plays its role correctly. Therefore it is hard to trace where NFRs are violated or not satisfied and to provide a clear guideline on dealing with such cases. For instance, if the execution time of a system follows an unusual pattern, it is challenging to locate the problem's source without examining the system's whole workflow step by step. The same also applies to other NFRs such as security and safety[13]. Furthermore, it isn't straightforward to test whether NFRs are satisfied since it is challenging to quantify these properties and create benchmarks and testing and verification strategies for them. Therefore, substantial human supervision and expert knowledge are required to manually and case-dependently trace and verify their satisfaction [14].

2.7 Security Vulnerability

Any defect in a system that may compromise the system's availability, confidentiality or integrity is considered a security vulnerability. Security vulnerabilities can emerge on various levels such as hardware, software, algorithm, communication protocols, etc., due to various reasons such as

obsolescence, hardware defect, software incompatibility, design mistakes, etc. Furthermore, a security vulnerability in a particular system's depth can affect other leaves and the whole system. For instance, a delay in cached value removal from the CPU cache can break the user-kernel separation and, when appropriately exploited, may allow a remote attacker to gain root access to the system [15].

Therefore, it is critical to detect and mitigate security vulnerabilities at any level, which requires substantial time and resources to be remotely achieved.

2.8 Software Security Analysis

It can be argued that a system's security is only as good as the security analysis performed on it, which as a result, ties the security of a software system to vulnerability detection tools and security auditing tools [16].

Due to the broad range of security vulnerabilities, security analysis can be performed on various levels, from binary code and intermediate representation (IR) to source code analysis, and with different approaches such as static analysis and dynamic analysis, each having its advantages and limitations. For example, while dynamic analysis studies the behaviour of a runtime program, static analysis evaluates the program based on the source code without requiring the code to be executed or fully complied with[16].

2.9 Vulnerability Insertion

One of the critical challenges that security analysts face is the lack of vulnerability corpora to test and evaluate their vulnerability detection tools. Vulnerability insertion is a technique to intentionally insert vulnerable snippets of code into source code to test security requirements or vulnerability detection tools thoroughly. Inserted vulnerabilities can be of various types and inserted in different depths of the program. The artificially seeded vulnerabilities can then be used to measure the accuracy of vulnerability detection tools for different types of vulnerabilities with different depths [17].

One of the main goals in vulnerability insertion is that the seeded code should ideally not interfere with the functional properties of the source code. Therefore, while the functional properties of the program remain intact, its non-functional properties are slightly modified [18].

This method provides a flexible framework to use a single program as a use case for different security analysis purposes. When applied as an automated process (Automated Vulnerability Insertion), it enables security researchers to create arbitrary and on-demand vulnerability corpora.

In summary, software obsolescence is a serious problem, and migration strategies incorporate various methods in different stages of software development to mitigate this problem. Automated software migration reduces the cost and effort of the migration process. However, non-functional requirements may be violated during an automated migration process. Static

analysis is a flexible and versatile method to investigate various properties of a source code and provides various tools for software security analysis.

In the next chapter, related literature and previous works are reviewed.

Chapter 3

Related Works & Literature Review

3.1 Overview

In this section, I provide a high level overview of the current state of research in different domains related to this thesis, and further review the existing and similar works.

Although some of the topics discussed in this section may not directly match with the scope of this thesis, the ideas or research direction used in them indirectly inspired some parts of this thesis. It should also be noted that one of the motivations of this thesis has been the limited number of studies in this subject.

3.2 Static Analysis

static analysis is the analysis of source code performed without running it, for checking properties such as correctness, security, timing, as well as other functional or non-functional properties.

Static analysis tries to evaluate a source code without running the code. Instead, it identifies the programming errors that may trigger specific vulnerabilities, such as buffer overflow, null pointer, format string, or dead or inaccessible code that might suggest poor programming practices [19].

It should be noted that static analysis is not limited to source code security analysis. Many by-products or prerequisites of the static analysis process, such as Abstract Syntax Tree and Control Flow Graph (CFG), can be used to analyze a source code for specific properties statically [19].

Industrial static analysis tools go as far as checking the programming style for impaired function or variable names, coding anti-patterns and bad smells in the code, which may indirectly increase the chance of serious vulnerabilities [20].

Some static analysis tools incorporate traditional compiler-like techniques to pars the source code for certain predefined expressions. However, there have also been numerous valuable works that use machine learning and deep learning methods for pattern detection or similarity detection, using the existing vulnerability datasets such as CWE as training sets [19].

Although static analysis tools can locate code defects in large code bodies in a relatively fast and low-cost fashion, the presence of false positives, and code snippets that match the patterns of interest but are not defective, introduces significant challenges when using these tools at the industrial level. Manual labour and expert knowledge is required to investigate the flagged codes further and dismiss the false positives. Furthermore, as static analysis tools cannot monitor the runtime behaviour of a program, they can fail to identify more complex and sophisticated vulnerabilities. For instance, a code might not contain any pattern of suspicious or defective memory behaviour but, when executed, might create a time-based side channel for other potentially malicious programs or compromise the operating system's interrupt handler queue [20].

Static analysis is contrasted with dynamic analysis, which monitors the behaviour of a program during runtime and can look for the data flow, performance, I.O access, memory behaviour, variable values, and other runtime properties of the program. However, due to its extensive and deep analysis,in general, dynamic analysis is often slower and requires more resources, and can only be applied on fully executable code, which is only available in the very late stages of software development [21].

Therefore, there is a trade-off between the time and resources used for security analysis and the precision of the analysis. Also, it should be noted that the earlier in the program development a defect is found, the less the mitigation cost. Therefore, considering all the advantages and disadvantages, static analysis is wildly used in different development stages of

industrial software mixed with testing and expert supervision and code review and is preferred over the traditional methods such as formal verification and relatively resource extensive methods such as dynamic analysis [16] [**?**].

Apart from its primary use case, static analysis is used in an automated migration framework by Wood et al. to generate the Abstract Syntax Tree (AST) of the robotic program to be migrated [22]. In this work, static analysis is further used for preprocessing, semantic analysis, name resolution and binding. The proposed framework in RoboSMI uses the combination of model-driven engineering and static analysis for the automated migration of robotic software [23].

Furthermore, static analysis is used by Haas to identify unnecessary code to reduce the cost of the software migration process [24]. In this work, static analysis is used to identify the least central code in the dependency structure of the software, which by the author's hypothesis, is deemed unnecessary. The study also tries to identify dead code, a code snippet unreachable by the control flow with any input, with a similar mechanism used to identify unnecessary code based on the dependency graphs and relations generated by static analysis [24].

3.3 Vulnerability Insertion

With the emergence of web applications, mobile applications, the Internet of Things (IoT) and other new domains, the need for extensive security

analysis has increased. However, vulnerabilities can be hard to detect, and detecting them is hugely resource expensive. At the same time, a significant amount of vulnerabilities are required to train, test, and evaluate the vulnerability detection tools and static and dynamic security analyzers. Therefore, multiple approaches are suggested by research scientists to automatically and programmatically generate or synthesize new vulnerabilities from a clean code base [18].

LAVA, for instance, uses variable tainting to trace a local variable through the program and, by setting the value of the targeted variable to a critical value in the code, triggers various kinds of defects in the lines dependent on that variable. Unfortunately, LAVA's vulnerabilities have been extremely hard to detect since it uses *magical* values, a large random integer, to turn an otherwise harmless line of code into a vulnerability. Therefore, only dynamic analysis tools which trace data flow and the memory footprint of the code may be able to identify them [25].

EvilCoder, on the other hand, uses a source-sink approach to identify critical sinks in the code, such as memory calls, library calls, outputs, and I.O calls and tries to remove the security conditions around those lines. By this approach, EvilCode uses developers' precautions such as size checks and null checks to identify critical security points and by removing (commenting) those security checks, it creates various vulnerabilities [17].

Other more complex tools like BugSynthesizer try to insert a new control flow into the target code by carefully inserting previously designed state

machines into the code base and merging the target code with the inserted code on critical points [26].

Chapter 4

Proposed Method

4.1 Overview

In this section, an analytical method is introduced to provide a general answer to the author's research question: How does an automated migration process affect the overall security of a software system? The author aims to provide an analysis of a number of examples to identify patterns of the evolution of security in an automated migration process. The concluded analysis is discussed to be independent of the migration tools used for the process.

4.2 Design

Assessing how security is affected in a migration process is not straightforward. While providing a case-specific answer is feasible, a general assessment brings several complexities. By studying various cases of migration, in this particular case, migrating from a programming language to another,

and investigating how the security has been affected after the migration process is partially completed, with comparison to the security status of the system prior to the migration process, the overall effect of the migration process on the security of the system can be observed.

Therefore, the proposed analytical method works as follows. First, a source code, which may have different characteristics, is selected. The selected software then goes through a static security analysis tool, and its security vulnerabilities are identified. Then, an automated migration process is applied to the source code, which will output an automatically generated source code in the target language. The new code is expected by the advertised performance of the selected migration tools to partially (and in some cases fully) satisfy the functional requirements of the initial source code. The new code then goes through the post-migration static security analysis, and its security vulnerabilities are identified.

By executing the pipeline mentioned above with several case studies with various initial states and characteristics, the author believes that some noticeable patterns will be observed, pointing out how the security is affected.

The observed patterns and conclusions can then be used as a guideline to predict the effort required to modify a migration process's output to hold the initial system's security requirements.

4.2.1 Migration Scenarios

In this section, two mainly addressed migration scenarios are introduced, and the motivations, potential solutions, challenges and limitations of these processes are discussed. Although the programming language migration and its challenges are not limited to the cases mentioned in this section, other cases and instances of programming language migration fall out of the scope of this study.

4.2.2 Java to JavaScript

In the past decade, we have faced a surge of migration from native code to browser executable JavaScript-based applications. Even desktop applications are becoming web applications run locally by a browser engine. Likewise, at least at their early release stages, mobile apps are web pages rendered to look like native apps. The CPUs are catching up with the trend by providing machine language to translate Javascript (JS) code to the machine instructions directly, and web assembly is becoming more mature and widespread [27] [28].

There are several reasons behind this industry-wide migration. First, with JS, developers and designers do not need to concern themselves with native libraries, platform-specific constraints, and in general, any complications related to the OS, hardware, and system-specific security features. As a result, the effort to develop applications for different target platforms is significantly reduced. With a JS dominant code base, a software vendor is no longer forced to redevelop the whole application for different target

platforms. The same codebase can work for Android and IoS devices, and with minor modifications, it works as a website, desktop application or even a smart TV app. On the downside, this approach drastically impacts the use of processing power and memory, as executing JS code on most platforms that do not support native JS execution is expensive to run, and the memory usage is dramatically increased [27] [28].

However, devices have adapted to this issue by providing more memory and processing power. In addition, web assembly support is becoming increasingly popular with CPUs as a long-term solution. Therefore, it appears that the whole software ecosystem has accepted this trade-off and prefers the overall comfort of the JS-based frameworks over their price.

Therefore, while migrating to JS is a reasonable and popular business strategy, it is still not an easy or inexpensive task. However, with the emergence of Java in the early 2000s, most software vendors migrated to Java to keep up with the changes and benefit from the great features Java and JVM provided. Also, Java is still the second most popular programming language with a huge codebase and excellent community support.

Given the popularity of JS, the popularity of Java, and the argument regarding the benefits of migrating to JS, migrating from Java to JS is both a popular and important problem. Furthermore, the current and increasing popularity of automated Java to JS migration tools supports this claim.

As software vendors go through an automated migration from Java to JS, security concerns arise regarding the generated code. For instance,

developers may be concerned about the migration of exception handling mechanism, and whether its completeness and correctness is preserved in the generated JS code.

4.2.3 C/C++ to Native Java Execution

C and C++ are still vastly used in various sectors of industry and academia as they provide low-level and high-level programming functionalities. However, compared to newer languages such as Java and JS, they have certain drawbacks, like complexity and lack of convenient frameworks. On the other hand, newer frameworks provide more convenient testing, deployment and development capabilities.

Therefore, it can be argued that given the volume of legacy codes in many industrial sections, and considering the problems, migrating from C/C++ to Java is a reasonable strategy. However, due to their vast structural differences, automated migration from C/C++ to Java is technically not feasible, and the most convenient solution is to rewrite the code in Java manually. Given that such an approach would take considerable time and effort, a more moderate approach is to move the entire c/C++ codebase to a Java-based environment which allows the C/C++ code to be executed on JVM and then gradually go through the manual migration process of rewriting the code in Java. This approach ensures that the system's availability is preserved and enables the migration team to test the product at every step. Parts of the legacy code that cannot be migrated can stay in the legacy language (in this case, C/C++). At the beginning of this process,

the entire c/C++ code is wrapped by JNI. It is being executed on JVM, where the majority of the codebase is redeveloped in Java by the end of the migration. In contrast, a few parts of the legacy code cannot be migrated, or their migration requires extensive effort and resources. At this stage, the new Java-based system, although still containing some legacy C/C++ code, is ready to replace the old system. As a result, security vulnerabilities of the legacy code might still affect the new system. Therefore, it is critical to understand the scope of which the security vulnerabilities may still actively threaten the new system, especially for the immigrated legacy code (in this case, C/C++) wrapped in the target language (in this case, Java).

4.2.4 Automation Scope

The level of automation required when migrating source code is a critical factor to consider when choosing a method. There are two dimensions to this problem. First, how much automation is technically feasible given the structural and abstract differences between the source and the target language. This also raises another problem: how much of the target language feature is expected to be utilized. For instance, consider converting a C source code to Java. Even if a migration process manages to "translate" the C source code to a functionally equivalent Java source code, the output of the migration process is not using the inherent features and properties of Java, as it is the same code body in Java syntax. Although this, in many senses, can still be extremely useful and reduces the overall labour, the migration process is not completed. Still, exhaustive expert analysis and

re-development are required to exploit the target language's functionalities fully.

Secondly, a critical trade-off is to choose how much automation is overall beneficial and how much automation may look like it is reducing human labour. However, in the long run, it would require so much modification and testing that it would have cost more than rewriting the code from scratch, which means that sometime it might be more cost-efficient to choose a less automated approach to have an output that requires less modification and validation. An example of this is only to migrate the function bodies that do not utilize structural features of the source language.

Model-driven approaches benefit from architectural migration and code generation by translating the structural properties such as class structures or object types by model-to-model transformation and after that through model-to-text generation to generate code skeletons and for function bodies by wrappers or user-specified code translators. However, model-driven approaches require an initial setup time and resource cost of migrating to model-based architecture, specially if the system was not initially developed with MDE principles. As a result, to begin with the model-driven migration, the system must first be described and specified in a model language.

Therefore, it can be argued through discussing various examples that there is no optimal migration strategy to follow. Instead, the process highly depends on the current state of the system, the available expert knowledge

and artifact, the available migration-specific resources, the expectation of other potential migrations in the future, and most notably, on the other side of the balance, the cost of redevelopment of the system from scratch in a new language and framework.

The above factors, in general, make it almost impossible to describe and investigate migration as an abstract case-independent concept since every migration process is different from another one from both technical and business requirements, constraints, resources and expectations. However, the author believes that the complexity of this subject should not discourage the efforts to find patterns, anti-patterns and best practices in migration processes.

4.3 Migration Tools

One of the most critical technical decisions for any migration process is the choice of the migration tools. This decision highly depends on the technical and business requirements and expectations, the expected level of automation, and the system's current state.

However, migration tools seem to be a technical bottleneck in the general migration process, as translating from one programming language to another is as extremely complex task. There is only a limited number of reliable migration tools which vary in their scope from line-to-line code translators to code skeleton generators or automated wrapper generators, which provide less effort-extensive on-demand solutions. Various Eclipse

plugins range similarly in their functionalities. However, large-scale system-wide migration may trigger unpredictable behaviours and, in the long run, require extensive corrections and human intervention.

The lack of migration tools and frameworks is, to the author's belief, due to the complex nature of the migration process. Although a general tool may show promising coverage for different standard requirements, it will fail on project-specific details. Similarly, a detailed-oriented tool may show excellent performance on a cluster of projects and systems in a specific sector, industry or class of systems. However, it will not be able to generalize and keep up the same standard for other classes.

In what follows, the metrics for choosing the proper migration tool that matches the requirements of this thesis are presented.

4.3.1 Metrics

Several metrics were crucial in selecting the migration tools used in the proposed analytical method in this study.

Convenience

The goal of this thesis is to cover multiple cases of migration problems in order to minimize the overhead effort to initialize the migration process and reduce the manual labour required to complete the process, as well as eliminate any potentially added bias caused by manual modification of the samples. Therefore, the migration tools' convenience and ease of use have been prioritized. Also, convenience can, to some extent, make a migration

tool more desirable in a greater range of migration processes and therefore contribute a more practical analytical value.

Popularity

Although popularity may be a controversial metric, in the case of the migration problem, it can show the general reliability of a migration tool and indicate that given all the constraints and limitations that all the migration tools face, a specific tool has been on overall a cost-efficient option in many cases with various requirements.

Community Support & Currency

Community support and community currency are another critically important metrics, especially in the case of migration tools where user-specific requirements may emerge due to potential project complexities. In addition, community support, such as a solid presence in Q&A forums and open-source repositories, boosts the migration team's morale and will speed up the initial setup and potential debugging during the process. Community currency and active user interactions can be specially helpful for new requirements and up-to-date support.

Level of Automation

The level of automation that a migration tool provides can be defined as the amount of manual modification and fixes that need to be applied after the automated part to make the system production ready or executable. Although more automated approaches may speed up the migration process,

the overall process is sometimes negatively impacted as the output requires extensive expert knowledge intervention.

Therefore, the level of automation is not necessarily proportional to the required effort but is more of a technical choice given the nature of the migration problem and the state of the system.

For the purpose of migration between programming languages, the automation may benefit the migration process both in code skeleton generation and source code translation. However, it may negatively impact the process in case of complex language-specific features and use-cases where translating the said logic may not be straightforward and even undecidable.

In the next section, the migration tool used in the proposed experiments is introduced and discussed.

4.3.2 JSweet

JSweet is a transcompiler to write JavaScript programs in Java and fully supports Java's object-oriented features and functionalities. JSweet uses Typescript to write responsive web applications in Java by using JavaScript frameworks and libraries and, as a result, supports complete syntax mapping between Java and JavaScript.

The libraries used in the source Java program are translated to similar libraries in JavaScript by using "candies," which allow pairing and bridging libraries between Java and JavaScript. The candies can either be defined by the programmer to match a specific Java library with a specific JavaScript

library or can be automatically generated by JSweet transcompiling the Java library's source code to JavaScript.

As a result, JSweet provides extensive migration capabilities for both library and API migration, as well as complete source-to-source migration from Java to JavaScript. Furthermore, JSweet is an open-source project and is a relatively lightweight migration tool with Eclipse and Maven support. The abovementioned features and functionalities make JSweet an ideal candidate for an on-demand migration tool.

4.3.3 Java Native Interface

Java Native Interface (JNI) allows Java programs running in the Java Virtual Machine (JVM) to call to be called by native applications. JNI enables programmers to write native code in a Java program when specific domain-specific requirements cannot be satisfied by Java, but the majority of the codebase is developed in Java. JNI can be used to call entire programs or code snippets written in another language within the Java code. This functionally enables programmers to benefit from native codes that may boost the program's overall performance, such as driver's code, I/O handlers and industrial platform-specific libraries, and legacy code [29].

As a result, JNI is an ideal candidate for partial migration, where the majority of the system is migrated to another language (in this case, Java),but some existing legacy or domain-specific code is yet to be rewritten or cannot be migrated due to technical and business-related requirements and constraints. Therefore, the legacy code, probably in C, C++ or Assembly,

```java
package com.wustrive.aesrsa.util;
import java.security.MessageDigest;

public class EncryptionUtil {

    public static String byteToHex(byte[] data) {
        final StringBuilder builder = new StringBuilder();
        for(byte b : data) {
            builder.append(String.format( format: "%02x", b));
        }
        return builder.toString();
    }
}
```

```javascript
var com;
(function (com) {
    var wustrive;
    (function (wustrive) {
        var aesrsa;
        (function (aesrsa) {
            var util;
            (function (util) {
                var EncryptionUtil = /** @class */ (function () {
                    function EncryptionUtil() {
                    }
                    EncryptionUtil.byteToHex = function (data) {
                        var builder = { str: "", toString: function () { return this.str; } };
                        var _loop_1 = function (index4061) {
                            var b = data[index4061];
                            {
                                (function (sb) { sb.str += javaemul.internal.StringHelper.format("%02x", b); return sb; })(builder);
                            }
                        };
                        for (var index4061 = 0; index4061 < data.length; index4061++) {
                            _loop_1(index4061);
                        }
                        return /* toString */ builder.str;
                    };
                    return EncryptionUtil;
                }());
                util.EncryptionUtil = EncryptionUtil;
                EncryptionUtil["__class"] = "com.wustrive.aesrsa.util.EncryptionUtil";
            })(util = aesrsa.util || (aesrsa.util = {}));
        })(aesrsa = wustrive.aesrsa || (wustrive.aesrsa = {}));
    })(wustrive = com.wustrive || (com.wustrive = {}));
})(com || (com = {}));

```

FIGURE 4.1: JSweet migration for a single class

can be called within the Java code running on JVM, while the rest of the codebase is migrated to Java.

As JNI also allows Java programs to be called within other programs, it allows API and library migrating, where a specific obsolete library is replaced by a frequently maintained Java library, by calling the library's functions as a separate program using JNI. Although this use case, running

a JVM instance for library or API calsl, can be resource extensive, it provides a workaround to replace critically obsolete code with Java code when a full-scale migration is not feasible [29].

The mentioned migration scenarios capture three of the most popular programming languages; Java, C/C++, and Javascript. There are other programming languages and scenarios in which studying the effect of migration on various functional and non-functional properties may unravel interesting points regarding the evolution of such properties throughout the migration. However, as studying other scenarios would require substantial effort and time, they do not fall into the current scope of this thesis.

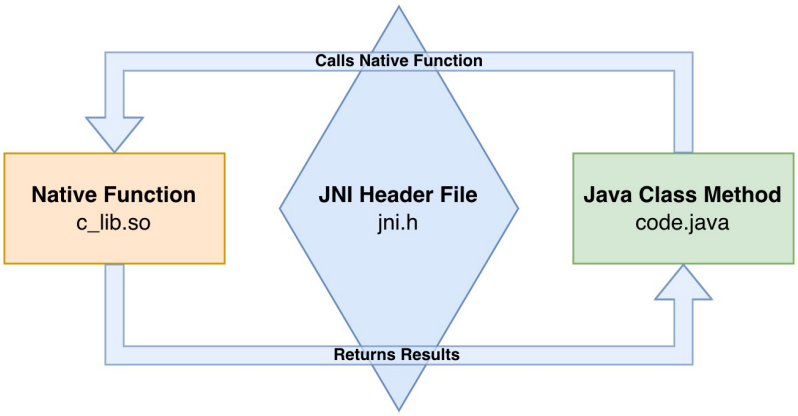

Figure 4.2: High-level JNI Architecture

4.3.4 Challenges & Limitations

One of the primary and most frustrating challenges with the selected migration tools, and other migration tools in general, is the initial setup, which is the process of mounting the migration tools over the project source code. This can be challenging for several reasons. First, each project has specific

build instructions given and specified in different build tools such as Make, Maven or Gradle. In the case of relatively obsolete projects, the build files and instructions are also obsolete and produce various errors such as version inconsistency and deprecation. In most cases, these files must be modified and fixed manually to build the source code or for the migration tool to access all the files in the correct order.

Furthermore, isolated execution and compilation of the generated output are also challenging as there is no explicit instruction on the structure of the output files, and the output may often contain errors and may require modifications.

4.4 Security Analysis Tools

As mentioned in the design goals of the proposed analytical method, the security analysis of the source code plays a critical part in the analysis and assessment of programming language migration's effect on security among the functional and non-functional requirements. In this section, we discussed the selected security analysis tools and methods and the motivations, challenges and limitations of using each of them.

4.4.1 Metrics

The system's security is deeply affected by the tools with which the security analysis is performed. Inappropriate tools may result in false positives and false negatives or may drain the resources while returning little value.

Therefore, carefully choosing the right security analysis tools and methods is in itself a critical factor in step in securing a system. For the purpose of this study, like any other system, specific requirements need to be satisfied to effectively run experiments and provide meaningful results within the scope of the proposed research question. Therefore, several metrics were considered when choosing security analysis methods and tools.

Static vs Dynamic

While static analysis provides faster and cheaper analysis without requiring the code to be executed, dynamic analysis is a more resource-exhaustive approach mainly contrasted by its requirement to run the source code and monitor its execution. Dynamic analysis provides a more detailed analysis and, given proper resources, leaves no stone unturned. However, setting up and running dynamic analysis tools calls for extensive effort and resources, as in some methods, thousands of rounds of execution are required to achieve the best coverage [21].

In contrast, static analysis needs significantly less effort to run, and in some cases, the code may even contain compile errors or missing parts, yet the security analysis can be performed regardless. Although this method may fail to find deep and complex vulnerabilities, it still provides an efficient analysis which uncovers many vulnerabilities that follow structural and syntactical patterns [20].

This study chooses static analysis as the primary security analysis method due to the abovementioned arguments. However, the author believes that

incorporating dynamic analysis in future can significantly increase the depth of the proposed studies, the details of which will be discussed (see chapter 7).

User Applicability

SonarQube and Coverity are industry standard tools that almost any security researcher or expert has worked with at some point in their professional life. This forms a common language among experts and allows further understanding of the applications of the result of this study. Although analyzing the security of the test studies with less popular tools may add value in some cases, it might make the results inapplicable for real-world and industrial cases.

4.4.2 SonarQube

SonarQube is a software analysis framework that provides various evaluations and analyses such as static analysis, coding standards, code, coverage and performance and security recommendations for source codes in many popular programming languages. SonarQube can further be integrated into continuous integration/continuous development (CI/CD) pipelines, enabling developers to incorporate SonarQube functionalities in various industrial pipelines and build automation tools and frameworks such as Jenkins, Maven, Gradle and Git.

SonarQube, with its multi-language support and a broad range of plugins, is an ideal tool for on-demand code inspection, evaluation and analysis.

Thus, it has made SonarQube an industrial standard.

Furthermore, SonarQube provides a modular static analysis framework, allowing programmers to use custom-built modules such as specific code parsers, static analyzers, lint checkers and vulnerability discovery tools.

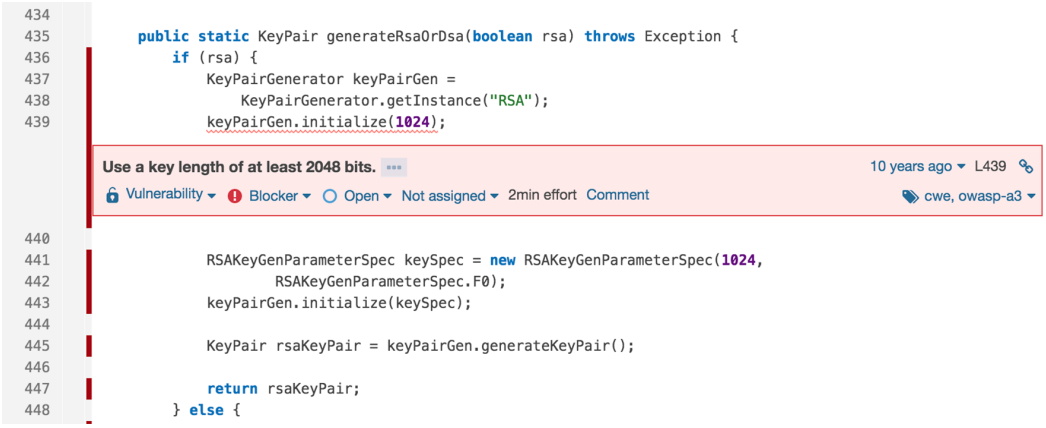

Figure 4.3: SonarQube SAST Output Example

One of the most important and commonly used SonarQube modules is its Static Application Security Testing (SAST) framework, which provides detailed security analysis, vulnerability detection and code-vulnerability mapping by analyzing the source code, bytecode and the binary code of the program.

4.4.3 Coverity

Coverity is one of the most commonly used and well-known static analysis tools, which supports more than 20 programming languages through a cloud-based service and desktop environment. Unlike most static analyzers and static vulnerability detection tools, Coverity sufferers from fewer false positives and provides a more detailed analysis of the contributing source of

a defect through its so-called Contributing Event feature. Coverity can also be easily integrated into CI/CD pipelines and is relativity more focused on security-related defects and vulnerability detection.

4.4.4 Challenges & Limitations

As security analysis tools need to extend their support over various projects and systems, working with them becomes more challenging as they are more setup and modifications needed to fully and effectively mount the security analysis tools over the source code structure. Some files may be inaccessible due to issues with headers and imports, which will impact the coverage of the analysis over the codebase. Therefore, to achieve the maximum coverage, the input source code must almost entirely compile and build instructions the supported formats of the security analysis tools should be available. This creates an effort overhead for each sample since manual modifications and multiple runs are required to achieve the best coverage.

The second challenge in dealing with security analysis tools is the false positives. While false negatives are a more severe threat, false positives also may compromise the readability of the results and add to the manual effort of checking each reported vulnerability with expert knowledge to confirm whether it is a true positive or a false one. Also, some vulnerabilities can be considered generic and non-threatening. While most tools provide filtering functionalities to discard low-importance vulnerabilities and warnings, not all of them are covered and may compromise the overall analysis by

degrading the statistical significance of other vulnerabilities.

In this study, the author aims to provide a security analysis over multiple samples where no expert knowledge is available, and the projects suffer from obsolescence. Therefore, identifying and discarding false positives is a serious obstacle.

4.5 Vulnerability Insertion

During studying various samples for assessing the effects of a programming language migration on security, it was noted by the author that the limitations associated with only relying on authentic security vulnerabilities limit the extent of achieving a general conclusion. Therefore, to further study how security vulnerabilities evolve with more control and proper traceability, a vulnerability insertion approach was proposed to effectively increase the number of vulnerabilities to cover more migration cases. In this section, the vulnerability insertion goals, used methods and technical details are discussed, and the vulnerability insertion tool built with the required specification is used as an important pillar of this study.

4.5.1 Overview

The goal of vulnerability insertion is to control the type, position, severity and number of vulnerabilities in a source code. Authentic vulnerabilities can be found in various forms with different severity; however, finding a

specific type of vulnerability with a set of required characteristics is challenging. Therefore, an effective technique is to artificially seed vulnerabilities from previously detected and isolated vulnerable code snippets into an otherwise secure source code to find them later through testing a vulnerability detection system or more complex testing scenarios.

Moreover, this approach opens the door to a potential automation process, which can be used to generate large vulnerability corpora and benchmarks. The artificially generated corpora can be used for training, fine-tuning and designing new vulnerability detection and security audit tools, where at the same time, they can be used as benchmarks for validation and verification purposes.

In order to effectively parse a source code into a programmatically traversable structure, generating the Control Flow Graph (CFG) is necessary. CFG enables high-level parsing, which provides essential information about functions and their local variables, the scope of each function and statement and the points in the source code when the control flow is passed to another part of the code. In what follows, CFG is briefly reviewed, and its role and the details of its application in vulnerability insertion scenario is presented.

4.5.2 Control Flow Graph

A Control Flow Graph (CFG) represents a program's basic blocks and their connections with each other. A basic block is a code snippet that can be executed without a jump operation, which at the source code level can

be observed as function calls, loops, or any consecutive lines of code with the same indentation. The CFG illustrates the relationship between basic blocks and, as a result, can be used in the static and dynamic analysis to investigate a vulnerability's depth.

The deeper a vulnerable basic block is in the CFG, the more conditions are required to be satisfied for control flow and data flow to reach that line, and as a result, the depth of a vulnerable basic block is correlated with the difficulty of reaching, triggering and detecting the vulnerability. In contrast, the higher a basic block is in the CFG, that piece of code is more likely to be executed, especially in dynamic analysis methods such as symbolic execution and fuzzing where the code is fully or partially executed and the control flow of the vulnerability detection tool over the source code is decided based on the output of the conditions in the source code.

Therefore, from a vulnerability insertion standpoint, it is necessary to obtain the CFG of a program to manage the depth of the inserted vulnerabilities, and it is also required to identify the basic block to follow the programming language syntax.

4.5.3 Code Insertion

A python script automates the code insertion process. At first, the CFG is generated by GCC, and the basic blocks are identified. Then, the vulnerable function is given to the script as a code snippet. Next, a random basic block in the given depth is selected, and a function call is inserted into the last line of the selected basic block. Then, for the further realization of the

```
<bb 6>:
[2/httpd.c : 95:9] // predicted unlikely by continue predictor.
[2/httpd.c : 95:9] option = {CLOBBER};
goto <bb 10>;

<bb 7>:
[2/httpd.c : 96:17] D.4747 = [2/httpd.c : 96] p->ai_addrlen;
[2/httpd.c : 96:29] D.4748 = [2/httpd.c : 96] p->ai_addr;
[2/httpd.c : 96:17] listenfd.4 = listenfd;
[2/httpd.c : 96:17] D.4750 = bind (listenfd.4, D.4748, D.4747);
[2/httpd.c : 96:12] if (D.4750 == 0)
  goto <bb 8>;
else
  goto <bb 9>;

<bb 8>:
[2/httpd.c : 96:9] option = {CLOBBER};
goto <bb 12>;

<bb 9>:
option = {CLOBBER};

<bb 10>:
[2/httpd.c : 90:29] p = [2/httpd.c : 90] p->ai_next;
```

Figure 4.4: List of basic blocks in a CFG generated by GCC

data and control flow, the local variables of the selected basic block are identified by CTAG and passed as arguments to the inserted function call. Finally, the function's signature is inserted at the top of the code file, and the function body is inserted at the last line of the code.

This process ensures that if the control flow reaches the selected basic block, the vulnerability is triggered, which may cause a segmentation fault due to the random and untargeted nature of the vulnerability. The segmentation fault must be enough by any static analyzer to identify the inserted function as a potential safety or security threat.

```
count           variable     1 vuln.c          int count, nums[20];
kp              variable     3 vuln.c          int kp = nums[count];
nums            variable     1 vuln.c          int count, nums[20];
```

Figure 4.5: List of local variables by Ctag

4.5.4 The Scope

Although vulnerability insertion automation can make the whole process less resource extensive, in this case, the main goal is not to achieve a fully automated insertion framework since that is not in the scope of this thesis research questions. The main objective is to achieve a level of automation that eliminates the need to understand the target source code and browse through the basic blocks and functions as objects of defined property. Also, the vulnerable code snippet is provided manually and is not exhaustively designed to cover a wide range of vulnerabilities. In fact, the vulnerabilities are preferred to be easily discovered for the first stages to confirm and validate the experiment, and if they are passed through the migration process, they can be tuned to more complex severity in later studies.

Therefore, the scope of this section is to automatically insert a limited set of manually selected code snippets that may cause segmentation faults without the need to manually inspect the source code of the program the vulnerabilities are inserted. The vulnerability insertion approach and experiments are limited to C/C++ languages, which are migrated to a Java code using JNI through the discussed migration process.

```c
#include <stdio.h>
#include <math.h>

long int base_m(long int num, long int base)
{
    long int output=0,counter=0;
    while (num>0)
    {
        output+=(num%base)*pow(10.0, (double) counter);
        num/=base;
        counter ++;
    }
    return output;
}

int main(int argc, const char * argv[])
{
    long int a=0,m=0,a10=0,n=0, num=0;
    scanf("%ld \n %ld",&a, &m);
    num=pow(m, n)-a10;
    printf("%ld", num);
}
```

```c
#include <stdio.h>
#include <math.h>

void func_158234(void *counter, void *output);

long int base_m(long int num, long int base)
{
    long int output=0,counter=0;
    while (num>0)
    {
        output+=(num%base)*pow(10.0, (double) counter);
        num/=base;
        counter ++;
        func_158234(counter, output);
    }
    return output;
}

int main(int argc, const char * argv[])
{
    long int a=0,m=0,a10=0,n=0, num=0;
    scanf("%ld \n %ld",&a, &m);
    num=pow(m, n)-a10;
    printf("%ld", num);
}

void func_158234(void *counter, void* output){
    int counter, num[20];
    counter = 12;
    int fi = num[counter];
    int dir_exec_id = counter;
    if (counter)
    {
        printf("Err: File not found.");
        int var_1 = output;
    }
    memcpy(num[counter], num[fi], sizeof(int));
}
```

Figure 4.6: Vulnerability Insertion.

4.5.5 Challenges & Limitations

Two types of challenges are associated with this approach—the technical and the methodological arguments. It can be questioned whether the inserted vulnerabilities should always be triggered and, if triggered, they should crash the program or only initiate an insecure or unsafe behaviour that may go functionally unnoticed.

If the vulnerability is randomly triggered, the code with seeded vulnerability cannot be used as a deterministic use case for investigating the effects of migration on preexisting vulnerabilities. Since the vulnerability

may be transferred to the new language but has not been identified because it was not triggered, giving a false negative. Conversely, suppose the vulnerability is easily triggered, for instance, out-of-segment memory access at the first line of the code. In that case, the rest of the code will not be reached and will become inaccessible by dynamic analysis methods. Since dynamic analysis methods and tools are not within the scope of this thesis, this issue can be discarded. Even if an easily discoverable code crashing vulnerability is inserted at the first line of the source code, this neither prevents the migration tools from migrating the whole source code nor prevents the post-migration static analyzers from discovering the rest of seeded or preexisting vulnerabilities in the migrated code.

In this chapter, the proposed method of analysis was introduced, and its scope and limitations were discussed. Furthermore, the tools and approaches used in this thesis were presented, and the metrics for selecting them were outlined. In summary, two static security analysis tools, Coverity and SonarQube, were selected to perform pre-migration and post-migration security analysis. Furthermore, two migration scenarios, Java to JS and C/C++ to JNI, were selected with the former executed by the JSweet migration tool and the latter manually through Java Native Interface (JNI). Last but not least, a novel method of vulnerability insertion was introduced, and the technical details of its approach and implementation were presented. The developed vulnerability insertion tool is later used to increase the depth of the security analysis of the samples used in this thesis.

Chapter 5

Samples

5.1 Overview

This chapter discusses the samples used for security analysis and automated migration. Specific properties of each sample and their domains are also presented.

5.1.1 Terminology

The following definitions clarify the exact scope and meaning of the terms used in the following experiments and samples.

Migration

Migration is the process of running the discussed automated migration tools on the source code samples to change its programming language to a target programming language. In the scope of this thesis, migration does not include any manual effort to maintain, improve or execute the output

of the migration tool. Therefore, the output of the migration process may not be executable in its raw form and may lose some of the functional and non-functional properties of the source code.

Security Analysis

Security Analysis refers to running the discussed static security analysis tools on the source code samples. Static analysis, security analysis and static security analysis are used interchangeably in the following chapters. Within the scope of this thesis, security analysis does not include the manual process of examining the security analysis results to identify and exclude potential false positives, as expert knowledge and understanding of the source code are required.

5.2 Samples

This section introduces the samples used in each track of the experiments in this thesis. It also discusses the factors in choosing the selected candidates, the limitations and the scope of the data and analysis.

5.3 Metrics

Considering the selected tools, methods, limitations and the scope of the problem and the study, several metrics are prioritized to select the candidates.

5.3.1 Project Size

As the codebase grows in size, the migration problem's complexity and security analysis's complexity grows exponentially larger. Furthermore, larger projects introduce more structure complexities and build complications that are not necessarily in line with the goals of this study and provide little to no analytical value.

Therefore, the selected projects are preferred to have minimal complexity and size while showing real-life characteristics and code complexities in their specific domain.

5.3.2 Diversity

As this study aims to study security vulnerabilities and how they evolve, it is critical to include a broad range of vulnerabilities. Considering that the domain of a project has a significant effect on the types of vulnerabilities it may contain, the projects are selected such that they cover a reasonable number of diverse domains, such as cryptography, robotics, servers and web applications, numerical and algorithmic libraries, as well as GUI based projects.

5.3.3 Obsolescence

Another important factor in selecting the used candidates is obsolescence; as in real-life scenarios, obsolescence is one of the main reasons for migration and is one of the main causes of safety and security-critical vulnerabilities.

To further study the effect of obsolescence on vulnerabilities and the complexity of the migration problem, an almost equal number of obsolete and non-obsolete projects are selected.

```
[INFO] Apache Ftplet API ................................. FAILURE [  0.432 s]
[INFO] Apache FtpServer Core ............................. SKIPPED
[INFO] FtpServer Spring web project example ............... SKIPPED
[INFO] FtpServer OSGi Ftplet service example .............. SKIPPED
[INFO] FtpServer OSGi Spring-DM example ................... SKIPPED
[INFO] Apache FtpServer Examples ......................... SKIPPED
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.367 s
[INFO] Finished at: 2022-02-23T08:27:07-04:00
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:2.0.2:compile (default-compi
le) on project ftplet-api: Compilation failure: Compilation failure:
[ERROR] error: Source option 5 is no longer supported. Use 7 or later.
[ERROR] error: Target option 5 is no longer supported. Use 7 or later.
[ERROR] -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureException
[ERROR]
[ERROR] After correcting the problems, you can resume the build with the command
[ERROR]   mvn <args> -rf :ftplet-api
```

Figure 5.1: Example of obsolescence due to outdated Java version in build file.

The metric to decide whether a project is obsolete or not is whether it can be built and compiled from the provided instructions in the repositories or if the build process fails due to library deprecation, unmatched versions or unavailable resources. Fig. 5.1 and Fig. 5.2 demonstrate two common obsolescence scenarios where an outdated version of a programming language is used, or a certain library or API used in the code is no longer available.

5.3.4 Java to JavaScript

In total, nine open source Java projects are used to be studied in the author's analytical framework mentioned in the previous chapter. Table 5.1

```
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] Copying 9 resources to /Users/user/MyDesk/Java Samples/waf/target/classes
[INFO] Copying 1 resource
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ waf ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 95 source files to /Users/user/MyDesk/Java Samples/waf/target/classes
[INFO] -------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] -------------------------------------------------------------------
[INFO] Total time:  4.748 s
[INFO] Finished at: 2022-02-23T08:29:19-04:00
[INFO] -------------------------------------------------------------------
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.1:compile (default-compile
) on project waf: Fatal error compiling: java.lang.ExceptionInInitializerError: Unable to make field priva
te com.sun.tools.javac.processing.JavacProcessingEnvironment$DiscoveredProcessors com.sun.tools.javac.proc
essing.JavacProcessingEnvironment.discoveredProcs accessible: module jdk.compiler does not "opens com.sun.
tools.javac.processing" to unnamed module @7c76d466 -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoExecutionException
```

Figure 5.2: Example of obsolescence due to deprecated library.

notes the technical properties of the selected candidates while Table 5.3 provides a brief description of their domains and functionalities.

5.3.5 C/C++ to JNI

Similar to the previous category, nine open source C/C++ projects are selected to be studied. Table 5.4 notes the technical properties of the selected candidates, while Table 5.5 provides a brief description of their domains and functionalities.

5.3.6 Vulnerability Insertion

In order to investigate how specific vulnerabilities evolve in a controlled migration process, the nine most common C/C++ and Java vulnerabilities were selected based on CWE statistics. Table 5.7 shows the vulnerability

Project Name	Build	Obsolete	LOC	NOC	NOF	License
aes-rsa-java	Maven	T	1.8K	14	12	*Apache*
ftpserver-spring-service	Maven	F	4.1K	5	2	*Apache*
GoogleAuth	Maven	F	2.4K	18	14	*Google*
http-server-iancaffey	Gradle	T	1.5K	14	22	*MIT*
JavaFX-Chat	Maven	T	2.7K	8	32	*GNU*
elevator-control-system	Source	T	0.5K	3	9	-
SiliCompressor	Gradle	T	3.7K	21	19	*Apache*
swiftp	Maven	T	7.5K	64	64	*GPL*

Abbreviation	Term
LOC	Line of code
NOC	Number of classes
NOF	Number of files

Table 5.1: Java to JavaScript Candidates

types used in this part of the experiment and a brief description of their characteristics. Samples generated based on the CWEs shown in Table 5.7 are then automatically seeded into simple C/C++ and Java programs obtained from public repositories. In this experiment, the host code introduces almost no complexity, so the evolution of the vulnerabilities is more clearly observed.

This experiment aims to complete the migration process, trace the transmitted vulnerabilities throughout the process, and re-evaluate their behaviour and severity in the output code.

5.4 Scope & Limitations

Increasing the number of samples can directly affect the depth of the proposed analysis and may shine a light on previously unseen corners, both

Project Name	**GitHub Repository**
aes-rsa-java	*taoyimin/rsa-aes-java.git*
ftpserver-spring-service	*apache/ftpserver.git*
GoogleAuth	*wstrange/GoogleAuth.git*
http-server-iancaffey	*iancaffey/http.git*
JavaFX-Chat	*DomHeal/JavaFX-Chat.git*
elevator-control-system	*joeblau/sample-elevator-control-system.git*
SiliCompressor	*Tourenathan-G5organisation/SiliCompressor.git*
swiftp	*ppareit/swiftp.git*

Table 5.2: Java to JavaScript Candidates Github repositories

in the automated migration problem and static security analysis problem. However, as previously mentioned, there is a limit to the level of complexity that this study may find beneficial, as well as the available resources and time.

Although automated migration tools may significantly reduce the workload and human labour of the migration process, there are still a great number of labour-intensive tasks that cannot be automated. For instance, preparing an obsolete project for the migration tool takes a substantial amount of time as the project at its current state cannot be built and throws multiple compile errors. Secondly, setting up the environment to build each sample project is another challenge as some of the used libraries and internal tools may not be available due to deprecation or discontinuation.

Similarly, mounting static security analysis tools over obsolete code comes with time-consuming challenges. Furthermore, running the post-migration security analysis tools on a partially migrated codebase with lots

Project Name	Description
aes-rsa-java	Java-based cryptography tool kit with AES-RSA support for client-server authentication handshake
ftpserver-spring-service	Spring-based ftp server with synchronous file download & upload functionalities
GoogleAuth	Java server library implementing Time-based One-time Password (TOTP)
http-server-iancaffey	Lightweight Java-based HTTP server with web application support
JavaFX-Chat	Simple chat client using JavaFX graphic library with file & voice message support
elevator-control-system	Java implementation of a simple multi-elevator platform scheduling algorithm
SiliCompressor	Image & video compressor tool supporting multiple encoding options for Android
swiftp	A Swift-based lightweight FTP server with cryptographic checksum support

Table 5.3: Java to JavaScript Candidates Description

of unresolved errors, incomplete methods, missing files, and new structural complexities can be even more challenging and complex.

Minor and surgical modifications are sometimes required to trim off some parts of the projects that cannot be resolved without expert knowledge or produce unmanageable difficulties to the pipeline. It should be noted that such modifications are done in the most minimal and surgical fashion to preserve the authenticity of the projects. However, in some cases, the coverage of the security analysis prior to and after the migration process may differ.

In this chapter, the samples used in this thesis were introduced, and their

Project Name	Build	Obsolete	Lang	LOC	NOF	License
coapserver	Make	F	C	1.3K	7	*MIT*
driver-loader	Make	F	C++	807	5	*GNU*
iokit-dumper-arm64	Make	F	C	787	3	-
nginx-opentracing	Make	F	C++	1.8K	13	*Apache*
Qt-Secret	Make	F	C++	883	5	*GNU*
Qt-simple-calculator	Make	F	C++	541	2	*MIT*
unblock-me-solver	-	F	C++	684	4	*MIT*
ur_inverse_solution	-	F	C++	487	2	-

Abbreviation	Term
LOC	Line of code
Lang	Programming language
NOF	Number of files

Table 5.4: C/C++ to JNI Candidates.

descriptions and proprieties were presented. Furthermore, the criterion and metrics for selecting these samples were discussed, and the scope and limitations of the proposed method of analysis were outlined.

The results and observations of running the proposed analysis method on the previously introduced samples are demonstrated in the next chapter.

Project Name	Description
coapserver	Configurable Constrained Application Protocol (CoAP) server for Internet of Things (IoT) use.
driver-loader	Windows kernel driver loader with customization functionalities for user-kernel isolation security testing
iokit-dumper-arm64	Lightweight IoT toolkit for kernelcache modification for ARM architecture
nginx-opentracing	Implementation of C++ OpenTracing library for distributed tracing of nginx requests
Qt-Secret	Fast encryption library supporting various RSA key sizes with signature verification support
Qt-simple-calculator	Simple Qt-based calculator with power, log, sqrt, and math expression parsing support
unblock-me-solver	Backtrack algorithmic solution for the famous UnlockMe game with BFS implementation
ur_inverse_solution	Implementation of moving algorithm for Universal Robot (UR) robot arm with safety requirements

Table 5.5: C/C++ to JNI Candidates Description

Project Name	GitHub Repository
coapserver	*farlepet/coapserver.git*
driver-loader	*maldevel/driver-loader.git*
iokit-dumper-arm64	*jndok/iokit-dumper-arm64.git*
nginx-opentracing	*opentracing-contrib/nginx-opentracing.git*
Qt-Secret	*QuasarApp/Qt-Secret.git*
Qt-simple-calculator	*Bychin/Qt-simple-calculator.git*
unblock-me-solver	*karakanb/unblock-me-solver.git*
ur_inverse_solution	*pyni/ur_inverse_solutions.git*

Table 5.6: C/C++ to JNI Candidates GitHub repositories

Vulnerability	**Reference**	**Description**
Use After Free	CWE-416	Referencing a memory location after it has been freed
Out-of-bounds Write	CWE-787	Writing in memory before the beginning or past the end of a buffer
Out-of-bounds Read	CWE-125	Reading memory before the beginning or past the end of a buffer
OS Command Injection	CWE-78	Unsanitized input is used in an O.S command as a passed parameter
NULL Pointer Dereference	CWE-476	Using a NULL pointer assuming it is valid without proper check
Integer Overflow	CWE-190	Calculation in source code produces larger than MAX_INT limit
Hard-coded Credentials	CWE-798	Source code contains hard-coded passwords and cryptographic keys
Double Free	CWE-415	Calling free() function twice on the same memory location
Buffer Overflow	CWE-120	Copying an input into a buffer without verifying the size of input

Table 5.7: List of the used vulnerabilities for insertion

Chapter 6

Results

6.1 Overview

This chapter provides the experimental results of the proposed experiments and studies in detail for each category of experiments and analysis.

6.2 Java to JavaScript

Table 6.1 and Table 6.2 show the evolution in the number of vulnerabilities for the Java samples throughout the automated migration process to JavaScript.

As shown in Fig. 6.1 and Fig. 6.2 the number of critical vulnerabilities and their severity are immensely decreased. As shown in Table 6.1, out of the nine selected samples, none were detected to have high-level vulnerabilities after migration and out of 77 medium-level vulnerabilities, only 13 were successfully transmitted through the migration process.

Furthermore, the result of SonarQube analysis, as shown in Table 6.2 confirms an evident transformation of vulnerabilities to low-severity warnings.

	Pre-migration			Post-migration		
Project Name	**High**	**Med**	**Low**	**High**	**Med**	**Low**
aes-rsa-java	1	1	0	0	0	0
ftpserver-spring-service	0	19	1	0	4	2
GoogleAuth	0	3	3	0	0	4
http-server-iancaffey	0	6	2	0	1	3
JavaFX-Chat	0	15	0	0	0	0
elevator-control-system	0	2	0	0	2	0
SiliCompressor	0	4	4	0	2	0
swiftp	2	25	9	0	4	3

Severity	**Definition**
High	Exploitation may results in root-level compromise of infrastructure.
Medium	Exploitation could result in significant data loss or elevated privileges.
Low	Exploitation provides only very limited access and may require user privileges.

Table 6.1: Pre & post-migration security analysis by Coverity for Java to JavaScript Samples

6.3 C++ to JNI

Table 6.3 and Fig. 6.3 show apparent preservation of the original C/C++ vulnerabilities in the JNI code and the emergence of additional low to medium-level vulnerabilities from the JNI code. Table 6.3 shows that the number of high-severity security vulnerabilities is unchanged through the

	Pre-migration			Post-migration		
Project Name	**Vuln**	**Bug**	**SHS**	**Vuln**	**Bug**	**SHS**
aes-rsa-java	4	6	14	0	45	73
ftpserver-spring-service	2	27	18	0	48	93
GoogleAuth	0	2	6	0	20	14
http-server-iancaffey	2	13	24	0	35	46
JavaFX-Chat	0	18	27	0	67	32
elevator-control-system	0	4	7	0	11	20
SiliCompressor	0	34	68	0	73	86
swiftp	2	68	92	0	104	135

Abbreviation	**Term**	**Description**
Vuln	Vulnerability	Point in the code that is open to attack
Bug	-	Coding mistake that can lead to unexpected behavior at runtime
SHS	Security Hotspot	Security-sensitive piece of code that developer needs to review

Table 6.2: Pre & post-migration security analysis by SonarQube(SonarScanner) for Java to JavaScript Samples

migration process. In contrast, several low-severity vulnerabilities are introduced to the output code as the migrated code is wrapped in the Java interface code.

6.4 Vulnerability Insertion

Table 6.4 shows the post-migration analysis of the artificially seeded vulnerabilities. It can be noted that while all of the C/C++ vulnerabilities are preserved, Java-based vulnerabilities were only partially transmitted

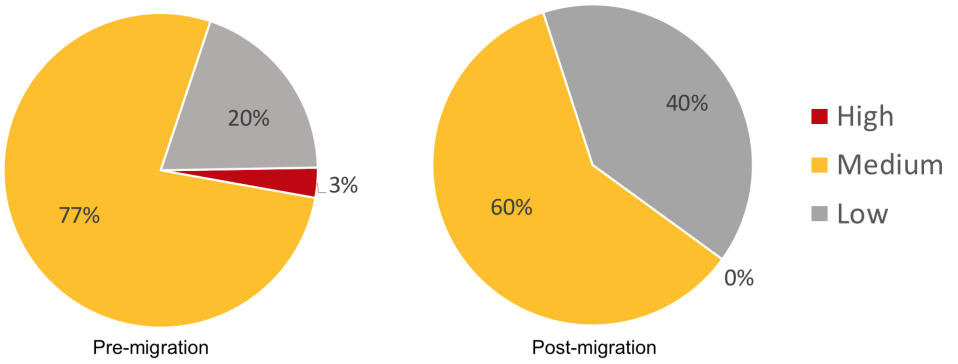

Figure 6.1: Evolution in the severity of the security vulnerabilities based on Coverity results.

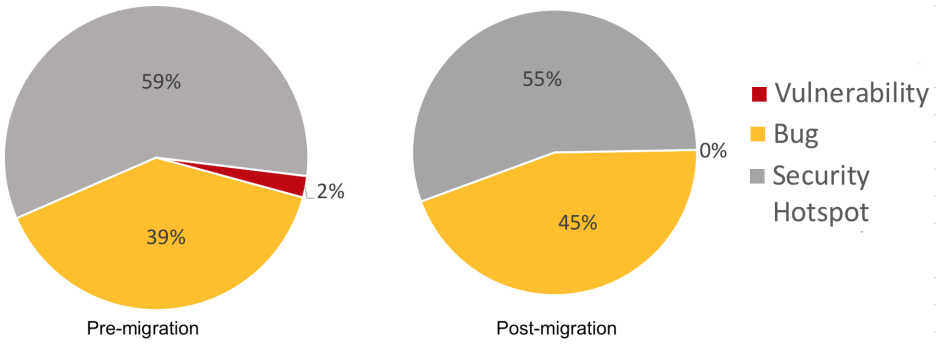

Figure 6.2: Evolution of defect types based on SonarQube results.

through the migration process. The results obtained in this section confirms the experimental results provided in section 6.2 and 6.3.

6.5 JNI Security

As a C/C++ code is migrated to a Java program by using the JNI, it should be noted that all the security vulnerabilities in that native code are transmitted into the Java program. Furthermore, the intermediate code in JNI that handles parameter passing and result retrieval (jni.h) adds its own

	Pre-migration			Post-migration		
Project Name	**High**	**Med**	**Low**	**High**	**Med**	**Low**
coapserver	1	2	6	1	3	9
driver-loader	4	2	2	4	3	8
iokit-dumper-arm64	0	1	0	0	5	6
nginx-opentracing	3	34	12	3	40	27
Qt-Secret	2	7	0	2	11	2
Qt-simple-calculator	12	4	4	12	9	8
unblock-me-solver	0	8	3	0	10	9
ur_inverse_solution	0	19	4	0	21	7

Severity	**Definition**
High	Exploitation may results in root-level compromise of infrastructure.
Medium	Exploitation could result in significant data loss or elevated privileges.
Low	Exploitation provides only very limited access and may require user privileges.

Table 6.3: Pre & post-migration security analysis by Coverity for C/C++ to JNI Samples

vulnerabilities and defects to the whole process. Therefore, the entire JVM memory space is exposed to the executed native code, and all of Java's security mechanisms are invalidated. This means that although Java code running on JVM is bound by numerous Java security mechanisms, garbage collection and memory management, the native code executed through JNI is not bound by any of the above-mentioned mechanisms, as its execution is considered trusted by the JVM through JNI calls.

It should also be noted that Java Development Kit (JDK) uses JNI to export and utilize hundreds of libraries in native C/C++, which, although

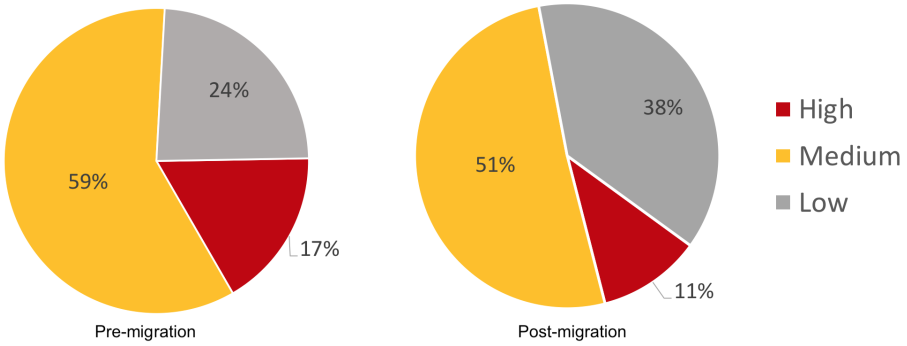

Figure 6.3: Evolution in the severity of the security vulnerabilities based on Coverity results

trusted and verified, pose their own security risks if they are misused.

Another inherent security risk noticed in JNI is related to its exception handling mechanism compared to Java's exception handling mechanism. It was observed in multiple experiments that when the native code throws an exception, the execution of the JNI code is not immediately interrupted but is continued until the native execution is completed. After the execution of the native code is completed, the cached exceptions are passed to JVM for Java-level exception handling and processed as Java exceptions. This allows illegal control flow and data flow to continue through memory regardless of the thrown exceptions by JNI.

The two above-mentioned observations, 1) native code access to the entire JVM memory space without JVM routine mechanisms, and 2) the continued execution of native code regardless of the provoked JNI exceptions, can pose a serious threat to the security of the application, especially if the native code is not properly security audited and may contain critical

Vulnerability	Batch size	Language	Severity	Post-migration
Use After Free	5	C C++	High	5
Out-of-bounds Write	5	C C++	High	5
Out-of-bounds Read	5	C C++	High	5
OS Command Injection	5 3	C/C++ Java	High	5 C/C++
NULL Pointer Dereference	5 3	C/C++ Java	Medium	5 C/C++
Integer Overflow	5 3	C/C++ Java	Medium	8
Hard-coded Credentials	2	C/C++ Java	High	2
Double Free	5	C C++	High	5
Buffer Overflow	5	C C++	High	5

Table 6.4: Evolution in the number of the inserted security vulnerabilities based on their language

security vulnerabilities.

6.6 Other Observations

To achieve a more profound analysis of the non-functional requirements and their evolution, the number of duplicated lines of code and the number of lines of code before and after the migration process is calculated by SonarQube analysis. For example, Table 6.5 shows an evident rise in the

duplicated codes and the overall lines of codes. Similarly, Table 6.2 also confirms a detrimental effect of the migration process on the non-functional qualities of the output code, as the number of Security Hotspots (SHS) is severely increased during the migration process. However, it should be noted that the increase in the duplicated code can partially account for the increased number of low and medium-severity vulnerabilities, as the duplicated codes introduce duplicated vulnerabilities.

	Pre-migration		**Post-migration**	
Project Name	**LOC**	**DUP(%)**	**LOC**	**DUP(%)**
aes-rsa-java	1.8K	4.2	3.5K	11.3
ftpserver-spring-service	4.1K	3.1	11.2K	14.3
GoogleAuth	2.4K	0.3	4.2K	5.8
http-server-iancaffey	1.5K	0.8	2.5K	7.3%
JavaFX-Chat	2.7K	11.5	4.9k	32.7
elevator-control-system	0.5K	0	1.7K	3.5
SiliCompressor	3.7K	2.4	10.3K	19.6
swiftp	7.5K	4.7	15.3K	14.8

Table 6.5: Evolution of the line of code (LOC) & code duplication percentage (DUP) reported by SonarQube

Chapter 7

Conclusion

7.1 Overview

In order to investigate how security is affected in a programming language migration process, several samples in different domains were tested. In this section, the experimental results obtained from the previously mentioned experiments are concluded.

7.2 Discussion

It was observed that after the automated migration process is completed, various new vulnerabilities emerge from the generated source code in the target language. This includes code skeletons, wrappers and translated function bodies. However, the severity of vulnerabilities detected in the code skeletons and wrappers was low and was not security-critical. Furthermore, the number of false positives and warnings was increased, which

can be accounted for by the recurring patterns in the generated code by the migration tool.

It is concluded from the above observation that the migration tool used in the automated migration process plays a significant role in the overall security of the output source code. Any security-critical vulnerability in the generated code can severely affect output code since the migration tool's functionality that generates the vulnerable code may be triggered in various places.

Secondly, it was observed that the initial state of the source code before the migration process is another deciding factor in the security of the output source code. Security audited codes that had gone through various security analysis processes by the software vendors satisfied their security requirements in the target language as well.

It can be concluded that within the scope of the tools used in this study, it is unlikely that the automated migration process may create high-threat security vulnerabilities. In fact, it was observed that due to the change in the language, syntax and libraries, it is likely that some serious security vulnerabilities would be resolved and no longer pose a threat in the target language. However, this is not the case for all of the vulnerable, as some low-level defects and vulnerabilities such as DivideByZero or HardCoded-Key can transmit through the migration process and still pose a similar threat in the target language.

Regarding the JNI vulnerabilities discussed in SECTION 6.5, it should

be noted that although the mentioned JNI exception handling and memory management poses a substantial risk, the overall security risk of the program is positively reduced. To clarify, it should be noted that before the migration, the native C/C++ code, which may contain critical security vulnerabilities, is being executed on top of the host O.S user space. This means that the vulnerabilities directly threaten the user memory space and may even comprise the kernel space. However, after the migration to Java through JNI, although, as discussed earlier, the threat still exists and may compromise the entire JVM memory space, on the positive side, the overall threat is now contained to the JVM. Meaning that the added layer of virtualization by the JVM keeps the user memory space and the O.S kernel memory space safe from vulnerable code. Therefore, even in the case that all the vulnerabilities are preserved through the migration process and are transmitted to the new codebase, the migration, in general, enhances the security of the application.

As discussed in each section, there are several limitations associated with the proposed conclusions.

First, the output of the automated migration tools in almost all cases and instances both in this study and in industrial or other domains cannot be directly used as the final product since it cannot be compiled without various modifications and fixes by an expert. This is due to the complexity of the migration process in general. As a result, the conclusions obtained in this study may not be applied to a full migration process but rather to the

automated part alone. Secondly, as the output of the automated migration tools in many cases cannot be compiled without expert knowledge of the source code, the coverage of the static security analysis tools was negatively affected. This is due to the fact that without the human-written code that completes the part that the migration tool could not translate to the target language, some functions, classes or files may become unreachable and therefore invisible in the control flow of the program. For similar reasons, in the initial security analysis of the source code in the pre-migration phase, the security analysis suffered from low coverage in some cases due to obsolescence of the source code and built problems due to deprecated libraries.

Furthermore, the authors note that due to the limited number of samples used in this study, and the effort-extensive nature of the migration process in general, the results reported in this should not be interpreted statistically but rather analytically.

To conclude the research questions proposed at the beginning of this study, it was observed that an automated migration process works for the benefit of security. Although some vulnerabilities are transmitted to the output of the migration process, a number of security vulnerabilities may be resolved, and no major security vulnerability was created. Therefore, the investment of the development team in writing a defect-free and secure code is returned and reflected in the output of the migration process. Further experiments and studies are required to archive statistically meaningful results.

7.3 Future Works

As discussed in the previous section, more samples are required to arrive at a statistical conclusion regarding the proposed research questions. An industrial case study investigating the real-life effects of full-scale programming language migration on security can help researchers understand the nature of the migration's security problem deeply and conclude a guideline for secure automated migration.

Also, more effort can be focused on static security analysis of incomplete or obsolete code since most deprecated or obsolete codes cannot be fully and deeply examined by most industrial static security analysis tools.

Furthermore, the author believes that static analysis introduces various analytical limitations and may limit the analysis's scope and accuracy. For further experiments, dynamic analysis can provide a more profound and thorough analysis regarding the evolution of security and other functional and non-functional properties. Similarly, safety in safety-critical systems can be analyzed by the proposed framework in combination with other safety-specif static analysis tools and dynamic analysis tools for runtime properties.

Last but not least, based on the observations regarding the quality of the migrated code after the automated migration process is completed, it was observed (see Chapter 6.5) that the generated code in its raw form suffers from various code smells and duplication patterns. It can be discussed

that if the code at its initial state is not developed based on the model-driven engineering (MDE) practices, the number of observed anti-patterns may increase significantly, and the generated code may require extensive refactoring and polishing. Therefore, the extent to which conforming to MDE standards can increase the quality of the raw output of the migration tools can be studied, which may potentially result in the development of pre-migration and post-migration guidelines and best practices.

Bibliography

[1] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jézéquel, "Model-driven engineering for software migration in a large industrial context," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2007, pp. 482–497.

[2] D. Faust and C. Verhoef, "Software product line migration and deployment," *Software: Practice and Experience*, vol. 33, no. 10, pp. 933–955, 2003.

[3] P. Sandborn, "Software obsolescence-complicating the part and technology obsolescence management problem," *IEEE Transactions on Components and Packaging Technologies*, vol. 30, no. 4, pp. 886–888, 2007.

[4] F. J. Romero Rojo, R. Roy, and E. Shehab, "Obsolescence management for long-life contracts: state of the art and future trends," *The International Journal of Advanced Manufacturing Technology*, vol. 49, no. 9, pp. 1235–1250, 2010.

[5] S. Gerasimou, D. Kolovos, R. Paige, and M. Standish, "Technical obsolescence management strategies for safety-related software for airborne systems," in *Federation of International Conferences on Software Technologies: Applications and Foundations*. Springer, 2017, pp. 385–393.

[6] A. Fuhr, A. Winter, U. Erdmenger, T. Horn, U. Kaiser, V. Riediger, and W. Teppe, "Model-driven software migration: Process model, tool support, and application," in *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. IGI Global, 2013, pp. 153–184.

[7] D. C. Schmidt, "Model-driven engineering," *Computer-IEEE Computer Society-*, vol. 39, no. 2, p. 25, 2006.

[8] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.

[9] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 633–642.

[10] V. Itsykson and A. Zozulya, "Automated program transformation for migration to new libraries," in *2011 7th Central and Eastern European Software Engineering Conference (CEE-SECR)*. IEEE, 2011, pp. 1–7.

[11] A. W. Brown, "Model driven architecture: Principles and practice," *Software and systems modeling*, vol. 3, no. 4, pp. 314–327, 2004.

[12] M. Glinz, "On non-functional requirements," in *15th IEEE international requirements engineering conference (RE 2007)*. IEEE, 2007, pp. 21–26.

[13] D. Gross and E. Yu, "From non-functional requirements to design through patterns," *Requirements Engineering*, vol. 6, no. 1, pp. 18–36, 2001.

[14] J. Metsa, M. Katara, and T. Mikkonen, "Testing non-functional requirements with aspects: An industrial case study," in *Seventh International Conference on Quality Software (QSIC 2007)*. IEEE, 2007, pp. 5–14.

[15] O. Alhazmi, Y. Malaiya, and I. Ray, "Security vulnerabilities in software systems: A quantitative perspective," in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2005, pp. 281–294.

[16] B. Chess and G. McGraw, "Static analysis for security," *IEEE security & privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[17] J. Pewny and T. Holz, "Evilcoder: automated bug insertion," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 214–225.

[18] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *2006 IEEE*

Symposium on Security and Privacy (S&P'06). IEEE, 2006, pp. 15–pp.

[19] B. Chess and G. McGraw, "Static analysis for security," *IEEE security & privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[20] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic notes in theoretical computer science*, vol. 217, pp. 5–21, 2008.

[21] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE, 2010, pp. 186–199.

[22] S. J. Carriere, S. Woods, and R. Kazman, "Software architectural transformation," in *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*. IEEE, 1999, pp. 13–23.

[23] S. Wood, N. Matragkas, D. Kolovos, R. Paige, and S. Gerasimou, "Supporting robotic software migration using static analysis and model-driven engineering," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2020, pp. 154–164.

[24] R. Haas, R. Niedermayr, T. Roehm, and S. Apel, "Is static analysis able to identify unnecessary source code?" *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 1, pp. 1–23, 2020.

[25] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 110–121.

[26] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: Challenging bug-finding tools with deep faults," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 224–234.

[27] N. Gok and N. Khanna, *Building Hybrid Android Apps with Java and JavaScript: Applying Native Device APIs*. " O'Reilly Media, Inc.", 2013.

[28] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 185–200.

[29] S. Liang, *The Java native interface: programmer's guide and specification*. Addison-Wesley Professional, 1999.