

MiniLib: A flow analysis–based approach for attack surface reduction through software debloating

Loukas Kopanias*, Panagiotis Sotiropoulos†, Nicholas Kolokotronis‡ and Costas Vassilakis§

Department of Informatics and Telecommunications

University of the Peloponnese

22131 Tripolis, Greece

Email: *dit17078@go.uop.gr, †panossot@uop.gr, ‡nkolok@uop.gr, §costas@uop.gr

Abstract—Software applications typically use libraries for the implementation of commonly used tasks. Each library encompasses an extensive collection of functionalities that cover a specific task area, such as interfacing with a database. However, while applications typically use a small subset of these functionalities, the unused ones are also bundled into the final distribution, due to the fact that the libraries are loaded and linked as indivisible objects. The presence of unused functionalities in the executable program increases its attack surface, since attackers may invoke code in these functionalities or exploit their vulnerabilities, using techniques such as stack smashing or buffer overflow. In this paper, we present MiniLib, an approach that removes from the final executable any unused functionalities that may be present in the libraries, reducing attack surface and thus enhancing security. The efficiency of MiniLib is validated through its application on applications drawn from the O-RAN 6G framework. Current findings indicate that the application of MiniLib may reduce the dependency-rooted application vulnerability exposure from 10.9% to 52.5%.

Index Terms—Software debloating; Attack surface; Security; Flow-based analysis; Vulnerabilities; Java bytecode; O-RAN; 6G.

I. INTRODUCTION

Libraries are nowadays indispensable elements of application development. They are bundled into applications to perform common tasks, accelerating development, promoting code portability and sharing, and enhancing software product quality, since the code realizing the functionalities supported by the library is tested and optimized [1].

Each library encompasses an extensive collection of functionalities that cover a specific task area, such as interfacing with the operating system, implementing a database, or performing mathematical computations. However, while applications typically use a limited subset of these functionalities, the unused ones are also bundled into the final distribution, too. This is owing to the fact that the linkage process, through which libraries are combined with the programmer-provided code into a single executable artifact, considers libraries as indivisible objects, and therefore if any piece of functionality

provided by a library is used in the program code, the whole library is bundled into the executable artifact; this applies also to the case when libraries are bundled statically into the executable, to facilitate deployment of microservices [2]. Additionally, at class-level, runtime mechanisms load the full set of methods and their implementations, while only a few of them may be actually needed. However, this practice leads to an increased attack surface for the application since:

- Extraneous library code (classes and/or methods) may contain vulnerabilities [3], [4], and attackers may use techniques such as stack smashing [5] or buffer overflow [6] to invoke the vulnerable code and exploit the vulnerabilities therein, leading to increased risk [7].
- When the program entails remote code execution (RCE) vulnerabilities [8], the extraneous code is made available to the code injected and executed by the attacker, further widening the functionality available to exploiters that have successfully performed an RCE attack. For instance, consider the case of a Java program uses a library (`jar` file) that references the `ProcessBuilder` class [9], but this class is not actually used: according to the loading practice described above, the class will be loaded, and will be then made available for invocation after a successful `log4j` vulnerability [3] exploitation.

At the same time, extraneous code increases the disk and memory footprints of executable artifacts.

In this paper, we present MiniLib, an approach that removes from the final executable any unused functionalities that may be present in the libraries, reducing attack surface and thus enhancing security. MiniLib adopts a flow analysis-based approach to identify reachable code, and once the reachable code has been identified, it removes all code that has been found not to be reachable. Reachability is determined using static code analysis ensuring efficiency, while the procedure follows transitive library dependencies, fully covering the program model. The MiniLib approach operates at the executable code level, allowing its use without necessitating the availability of source code or debugging information (which is required by other approaches, such as [10], [11]), and therefore extending the potential for its usage to additional cases, including commercial software where source code or debugging information is not available. Currently MiniLib is implemented for the

This work is partially funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or Smart Networks and Services Joint Undertaking. Neither the European Union nor the granting authority can be held responsible for them. The XTRUST-6G project (GA no. 101192749) is supported by the 6GSNS and its members.

Java language. The efficiency of MiniLib is validated through its application on applications drawn from the O-RAN 6G framework.

The rest of the paper is organized as follows: Section II overviews related work, and Section III introduces the proposed approach. Section IV presents and discusses the results of experiments, where the MiniLib approach is applied to a number of real-world cases. Finally, Section V discusses the limitations of the presented work, while Section VI concludes the paper and outlines future work.

II. RELATED WORK

Software bloating has been recognized as a serious issue in the software supply chain, leading to security issues and resource waste [12]. This has fueled research work targeting bloat analysis and debloating techniques. In this section, we overview work in this area.

Guoqing Xu et al. [13] describe the state of the art on bloat analysis and highlight some future directions in the field. In [14], the authors propose the Razor framework for Post-deployment software debloating which uses several control-flow heuristics to infer complementary code that is necessary to support user-expected functionalities. Furthermore, [15] introduces a debloating technique to improve memory usage when loops are concerned, determining which objects created within a loop can be reused and efficiently reusing such objects.

In [16], debloating is performed by combining static analysis and Machine Learning (ML), aiming to reduce attack surface. ML prediction aids static analysis by improving its precision, and static techniques augment ML models by providing mechanisms for identifying when a misprediction is truly an attack.

Agadakos et al. in [17] and [18], propose a binary debloating framework called Nibbler, a system that searches for unused functions within dynamically loaded libraries and removes them from the executable, achieving thus security benefits. Nibbler can be combined with additional security enhancements such as code re-randomization. In [19] D-Linker was proposed: a system that processes x86-64 applications identifying functions in its dynamically-linked libraries and that are not used and removes them, without necessitating access to source code. Furthermore, in [20], the authors use existing knowledge about the application, to minimize the maintenance effort required by the miniaturization activities.

In [21], Library Miniaturization Using Static and Dynamic Information is presented; this approach uses a web service-based architecture to collect data regarding the dynamic execution flow of applications, which are then combined with static analysis to determine how libraries should be minimized. Nevertheless, [21] aims minimize memory footprints, without considering security aspects. In [22], the JShrink framework is presented as a tool for software debloating, combining static reachability analysis with dynamic profiling analysis and type dependency analysis. In [23], the Chisel system is presented, which uses a model-based reinforcement learning approach to

accelerate the search for the reduced program and scale to large programs.

In [24], the authors introduce the Piece-Wise debloating framework, which combines compile-time and load-time approaches to systematically detect and automatically eliminate unused code from program memory. In this proposal, unused code is identified and removed by introducing a piece-wise compiler that (a) compiles code modules and (b) generates a dependency graph that keeps all compiler knowledge on dependent functions, enabling thus the identification of dynamically loaded functions that are needed by a program.

Césa Soto-Valero et al. in [25], propose the JDBL tool for debloating Java bytecode (JDBL). JDBL operates in three phases: during the coverage collection phase dynamic analysis is performed to gather the necessary information, while in the bytecode removal phase the redundant code is identified exploiting coverage data and eliminated. Finally, in the artifact validation phase, the correctness of the debloated artifact is verified. In [26], the authors propose a configuration-driven software debloating approach, according to which the libraries that are only needed for the realization of specific optional features are identified and mapped to relevant configuration directives. Afterwards, the build process is arranged so that each library is included in the executable only when the respective directives are set.

In [27], the authors present adaptive techniques for minimizing middleware memory footprint for distributed, real-time embedded systems, such as Conditional Compilation, Reflection and Dynamic Class Loading, Code Shrinkers, Code Obfuscators and Aspect-oriented Programming.

In [28] the authors elaborate on the trade-offs between software debloating and generality, quantifying the extent to which a debloated program behaves correctly also for inputs that were not present in the initial usage profile. The findings in this work indicate that in many cases code that would be required to process correctly certain inputs is removed, and to tackle this issue the authors propose and evaluate two augmentation methods.

Table I summarizes and compares the characteristics of MiniLib vs. representative works in the literature surveyed above.

III. THE MINILIB APPROACH

The MiniLib approach is designed to operate on bytecode level, rather than the source level, aiming to maximize (a) the amount of bloatware code that is removed from the executable and (b) the scope of its applicability, since it is able to operate without access to the application source code, which may not be available. Our prototype implementation access and manipulates elements that can be found inside Java archive (JAR) files. A JAR file is a compressed collection of compiled Java classes that can be executed in a Java Runtime Environment (JRE). After the compilation, the MiniLib processor is executed to process the newly created set of bytecode instructions, identifying and removing bloatware. Our approach does not remove directly redundant code from

TABLE I
CHARACTERISTICS OF MINILIB VS. OTHER WORKS

Approach	Mode of checking	Level of operation	Need for user input	Languages	Goal
<i>Razor</i> [14]	Dynamic	Binary code	✓	C, C++, Python	Security, footprint
Bhattacharya [15]	Static	Binary code	×	Java	Garbage collection
Porter [16]	Static & Dynamic	Source & Binary code	✓	Python	Security
<i>Nibler</i> [17] and [18]	Static	Binary code	×	Java	Security, footprint
<i>D-Linker</i> [19]	Static	Object code	×	ELF binaries	Security, footprint
<i>Chisel</i> [23]	Static & Dynamic	Source	✓	C	Security, footprint
<i>Piece-wise</i> [24]	Static	Binary	×	C	Security, footprint
<i>Minilib</i>	Static	Binary code	×	Java	Security, footprint

existing JAR files, but generates new class files and populates them with usable parts of bytecode.

In addition to executable debloating, MiniLib includes functionality to raise developer awareness regarding library-rooted vulnerabilities that still persist in the executable after the debloating operation. The MiniLib implementation extracts from the Common Vulnerabilities and Exposures database (<https://www.cve.org/>) class names and method definitions that entail vulnerabilities. Subsequently, during the minification phase, it either removes the vulnerable methods from the final executable file, or issues relevant warning in the log files, if some methods cannot be removed because they are needed in the application execution flow. In the following subsections, the core engine of the MiniLib architecture and the use of annotations to support multiple minified environments and Continuous Integration (CI) are discussed.

A. The Engine

The core MiniLib engine consists of four main components, the *File Finder*, the *Class Insider*, the *Tree Printer*, and the *Class Generator*. Fig. 1 depicts the overall architecture of the MiniLib system.

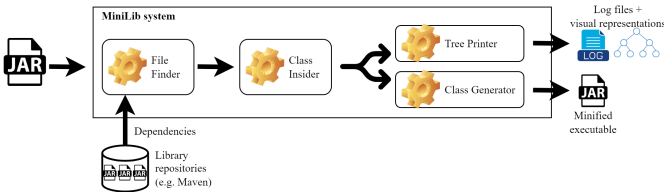


Fig. 1. The MiniLib architecture.

The File Finder module is responsible for scanning directories that store Maven dependencies, keeping the JAR files, and then extracting the bytecode classes. Before the minification process commences, it is important to ensure that the dependencies of the targeted project have been downloaded into the dependencies directory. If the source code is available, then this can be ensured through the `mvn compile` target; otherwise dependencies can be readily extracted from the application’s classpath.

Once class files are ensured, they are all loaded into a class pool, along with the executable JAR which is targeted for minification. The latter will be the starting point from

which the Class Insider will begin the execution tree traversal, in order to identify the complete set of methods that are called during application execution. The exact starting point of the execution trace is designated upon execution of the minification process.

In order to keep MiniLib simple and robust, at this phase of the implementation a single third-party library, Javassist (<https://www.javassist.org/>), is used for bytecode manipulation. Javassist provides a high-level interface to manipulate classes and fields at runtime, after the Java Virtual Machine (JVM) loads them. While Javassist provides functionality for source code modifications, this feature is not needed or used in our approach. Javassist can list the methods that are invoked from a specific of a method, and this feature is recursively called –starting from the program entry point– to build the complete method call tree. Notably, this Javassist functionality returns all the methods that are listed in the target method’s bytecode, not excluding methods that are unreachable, e.g. when they are placed inside an “if” block whose condition is never satisfied or after a conditional return statement which is always executed. In this sense, the method call tree may include methods that are not bound to be invoked, thus missing opportunities for further augmenting the effectiveness of the debloating procedure. This issue can be addressed by dynamically tracing program execution. This direction will be explored in our future work. When the recursive traversal of all invoked methods is complete, the method call tree contains exactly the classes and methods that are actually used during the execution of the application, while unused methods and classes will not be present in the tree.

Internally, the method call tree is represented using a generic (N-ary) tree, following the composite design pattern [29]. The composite pattern provides the flexibility to represent the nodes as individual objects. A node in the tree can represent a method, a constructor call, an interface, a super-class, an abstract class, or a field occurrence, and edges in the tree denote either containment (e.g. a method belongs to a class) or execution flow (e.g. a method `Class1.method1()` calls another method `Class2.method2()`).

Cache mechanisms were implemented to optimize workload balance and prevent unnecessary tree visits. Applications with many dependencies could generate large trees, and visiting the nodes multiple times would significantly degrade performance.

Each method call is stored into the tree only once, and its edge is created on the first occurrence. Multiple nodes representing the same element could potentially result in a finer tree representation (especially if distinct execution paths specific to call hierarchies were modeled), but would add more complexity.

Once the method call tree has been constructed, the two last modules of the Minilib system, namely the Tree Printer and the Class Generator are activated. The Tree Printer module traverses tree nodes and produces structured information both in textual and graphical format. Data visualization using a tree structure is performed using the GraphViz library (<https://graphviz.org/>). Tree nodes of different types (methods, classes, or interfaces) are denoted using different colors. The Tree Printer module extracts information from the CVE database, marking methods that are known to entail vulnerabilities, and inserts corresponding warnings into the log files that are created. In this way, developers and information security officers are made aware of potential security issues of the application.

Finally, the Class Generator module is used to create the final minified package. The Class Generator module traverses the method call tree, and for each class, method, or interface encountered, the corresponding elements are extracted from the class pool generated by the File Finder module and placed in the minified executable. The minified executable is packaged as a “fat JAR”, i.e. a self-contained JAR file including all elements needed for its execution, without any external dependencies.

Both the Tree Printer and the Class Generator modules utilize the visitor design pattern [30], to ensure tree traversal in a flexible, adaptive, and reusable way.

B. Annotation Parser

The MiniLib system allows the use of annotations to designate the mode of test method processing. In this context, two types of annotations can be used inside a unit test class. The first one is *class parsing*, where whole test classes are selected but each method is still processed separately, i.e. each method is parsed and minified individually. The second type of annotation is *selective parsing*, where the user can select which test methods should be extracted and used as entry points in the minification process, excluding those that are not needed. Annotations are essential when using Continuous Integrations (CI) tools to run sets of tests, and create multiple minified environments. If the application source code is not available, annotation functionality can be simulated through configuration files.

Annotation processing also serves the purpose of detecting method invocations that cannot be detected by static code analysis. Notably, this is the case of applications acting as servers, where the methods that are actually called are determined dynamically during runtime (e.g. after a web service invocation). For instance, in the Spring Framework, the mapping between URI paths and service method names is maintained in string format within a

`RequestMappingHandlerMapping` structure, and upon the reception of a request, the suitable service method name is retrieved from the structure and mapped to the method implementation, which is invoked dynamically. In these applications, a test class may be created which invokes all pertinent methods by explicitly referencing them (either from within a single test method or from within multiple methods) and the test class can be specified as the scanning entry point. Under these scheme, service classes will be included in the minified versions, and the same will also hold for their dependencies, as a result of recursive scanning.

IV. EVALUATION

In order to assess the effectiveness of the MiniLib approach, we conducted a set of experiments, in which MiniLib was applied on software programs in the O-RAN suite¹. The goal in this test is firstly to assess the level of attack surface reduction, due to removal of vulnerable code, and secondarily to minimize the final bytecode footprint. The following experimental procedure was performed for each software program within the testsuite:

- 1) The library-rooted vulnerabilities that were present in the original version of the testsuite application were scanned using the OWASP Dependency Check tool².
- 2) the methods that are involved in the vulnerability were identified, by examining the sources of the respective library and identifying the commits that fixed the particular vulnerability. At this phase, some of the vulnerabilities were found to relate to documentation or to languages not supported by the current MiniLib implementation (Kotlin, JavaScript or C++ (which is invoked through Java Native Interface)).
- 3) the application was compiled and minified.
- 4) the minified application was assessed to determine whether the vulnerabilities in the original code persist.

Since no exploits for the detected vulnerabilities were available, in order to actively verify whether the minified code is still vulnerable, the following method was used to estimate the presence of vulnerabilities in the minified code:

- 1) if *all methods* that were modified by the commits fixing the related vulnerability, then the vulnerability is considered to be fixed. Therefore, the impact score of the related CVE is deducted from the total vulnerability score of the application.
- 2) if *none of the methods* that were modified by the commits fixing the related vulnerability, then the vulnerability is deemed to still persist. In this case, the total vulnerability score of the application is not reduced.
- 3) if *some of the methods* that were modified by the commits fixing the related vulnerability, the vulnerability may either persist or not, depending on the method call sequence that should occur for the vulnerability to

¹<https://github.com/opennetworkinglab/smart5g-nonrtic-plt-ranpm>

²[urlhttps://owasp.org/www-project-dependency-check/](https://owasp.org/www-project-dependency-check/)

TABLE II
EXPERIMENTAL RESULTS FROM TESTSUITE APPLICATION MINIFICATION

Application	Dependencies	DR-AVS	MinR	MaxR
datafilecollector	jackson-databind, kafka-clients, logback-classic, logback-core, netty-codec-http, netty-common, netty-handler, reactor-netty, reactor-netty-http-brave, spring-aop, spring-core, spring-web, spring-webflux, spring-webmvc, swagger-ui	74.7	10.6 (14.2%)	39.2 (52.5%)
influxlogger	jackson-databind, json-path, json-smart, kafka-clients, logback-classic, logback-core, netty-codec-http, netty-common, netty-handler, okio, spring-aop, spring-core, spring-web, spring-webflux, spring-webmvc, swagger-ui, tomcat-embed-core	48.6	5.3 (10.9%)	15.3 (31.5%)
pmproducer	jackson-databind, json-path, json-smart, kafka-clients, logback-classic, logback-core, netty-codec-http, netty-common, netty-handler, spring-aop, spring-core, spring-web, spring-webflux, spring-webmvc, swagger-ui, tomcat-embed-core	48.6	5.3 (10.9%)	15.3 (31.5%)

manifest. In this case, we compute a minimum vulnerability score reduction equal to zero (the vulnerability fully persists in the minified executable) and a maximum vulnerability score reduction, equal to the CVSS score of the CVE (the vulnerability cannot manifest, due to the fact that the needed call chain has been broken).

The applications scanned during the experimental evaluation were *datafilecollector*, *influxlogger* and *pmproducer*, all drawn from the O-RAN 6G security suite. In the following paragraphs, we report on the results obtained, according to the method for attack surface reduction described above. Since all three testsuite applications operate as Spring apps, and therefore share many common dependencies, for conciseness purposes result presentation is organized as follows: Table II lists for each application (i) its dependencies that entail vulnerabilities, (ii) the overall dependency-rooted application vulnerability score (*DR-AVS*, which is computed as the sum of the CVSS scores of the CVEs), and (iii) the minimum and maximum vulnerability score reduction (*MinR* and *MaxR*), both as an absolute number and as a percentage of the initial score. Subsequently, table III lists for each library the CVEs it entails and for each of the CVEs (i) its CVSS score, (ii) the number of methods that were modified by the commits that fixed it (*#ModMed*) (iii) the number of these methods that persist in the minified executable (*#PerMed*), (iv) the minimum and maximum vulnerability score reduction achieved by the minification operation for the particular CVE (*MinRed* and *MaxRed*). In all cases, the impact of each vulnerability was retrieved from the CVE database, which provides a comprehensive score according to the Common Vulnerability Scoring System (CVSS).

Note that some dependencies share the same CVEs, due to transitive use of additional libraries: for instance, CVE 2024-38820 appears in dependencies *spring-webmvc* and *spring-aop*, since both of them transitively depend on the Spring Framework *DataBinder*, where the vulnerability corresponding to CVE 2024-38820 appears in. When a vulnerable dependency is included in an application through multiple paths, it is accounted only once in score calculations.

The results listed in table II demonstrate that through removal of vulnerable code, MiniLib is able to reduce the application vulnerability overall score by a ratio ranging from 10.9% to 52.5%. In applications with standard flow (i.e. method invocations are listed by the programmer within the

application code), this is accomplished by a simple invocation of the MiniLib executable, while in applications using more complex invocation logic (e.g. Spring-based applications), a set of tests needs to be developed. In all cases, the required effort is small (from a few minutes to one hour), and the security gains are considerable.

Regarding the application code size, this has been found to be reduced by a ratio ranging from 37% to 54%, reducing thus correspondingly the memory footprint.

V. LIMITATIONS

During the development phase, it was decided that the implementation of some extensions and optimizations could be deferred to a later stage, in order to create a proof-of-concept version and establish the appropriate foundation and structure of the code to support future revisions and enhancements. In this section, we discuss these aspects, highlighting areas for potential improvement.

A. Programming language

MiniLib considers the Java compilation and execution lifecycle, which has been studied and analyzed in depth. Similar lifecycle approaches are used in other languages, such as Kotlin or the .Net environment, which uses the MSIL (Microsoft Intermediate Language) [31]. Future versions of the MiniLib system will support additional languages.

B. Fields

Crawling inside a bytecode file looking for calling-method order cannot guarantee which fields are used. Depending on the recursive level the crawler is working on, it can become very difficult to determine which fields are used or will be used in certain method calls. We decided to keep all the fields that can be found inside a Java class file. This approach has a downside because it loads empty custom classes so not to break *ClassLoader*'s association between classes and unused fields. However, solving this problem would significantly increase the level of code complexity, reducing at the same time the footprint of the output.

C. Memory Usage

Every element is represented as an object in our graph. This causes some restrictions in terms of usability and scalability. Very large projects may take too long to be parsed and converted to a graph, and as a result, extensive memory

TABLE III
VULNERABLE DEPENDENCIES WITHIN THE TEST SUITE

Dependency	CVE	CVSS	#ModMed	#PerMed	MinRed	MaxRed
jackson-databind 2.15.3	2023-35116	4.7	13	12	0	4.7
kafka-clients 3.6.0	2024-31141	Not relevant; pertains to the configuration				
logback-classic 1.4.11	2023-6378	7.5	2	2	0	0
logback-core 1.4.11						
netty-codec-http 4.1.10.Final	2024-29025	5.3	13	0	5.3	5.3
netty-common 4.1.10.Final	2024-47535	5.5	4	4	0	0
netty-handler 4.1.10.Final	2025-24970	7.5	5	5	0	0
reactor-netty 1.0.22	2023-34054	7.5	1	1	0	0
	2023-34062	7.5	2	2	0	0
	2022-31684	4.3	37	16	0	4.3
reactor-netty-http-brave 1.1.13	2023-28360, 2022-47934 2022-47933, 2022-47932 2022-30334, 2021-22929 2020-8276	Not relevant; commits refer to native C++ code which is invoked using JNI				
spring-aop 6.1.1	2024-38820	5.3	46	18	0	5.3
spring-core 6.1.1						
spring-web 6.1.1	2024-38809	5.3	3	0	5.3	5.3
	2024-22243	3.4	4	3	0	3.4
	2024-22262	3.4	2	1	0	3.4
	2024-38820	5.3	46	18	0	5.3
spring-webflux 6.1.1	2024-38816	7.5	6	5	0	7.5
	2024-38820	5.3	46	18	0	5.3
spring-webmvc 6.1.1	2024-38816	7.5	6	5	0	7.5
	2024-38820	5.3	46	18	0	5.3
swagger-ui 4.15.5	2024-45801, 2024-47875 2024-48910	Not relevant; commits refer to JavaScript code				
json-path 2.7.0	2023-51074	5.3	4	4	0	0
okio 2.10.0	2023-3635	Not relevant; commits refer to Kotlin code				
tomcat-embed-core 10.1.16	2024-38286	7.5	6	6	0	0
json-smart-v2 2.5.0	2024-38286	7.5	1	1	0	0

usage may occur. Some optimizations in graph processing have already been implemented, including the use of references between entities and storing a single copy of class, method, and interface nodes, irrespectively of the number of times each of them is used; nevertheless, packages with many different calling signatures may still lead to the creation of a very large graph. Serial IDs, unique signatures, parent and child node reference lists are unavoidably included in the graph, as the minimal required of information, and appear to exhibit very limited opportunities for further optimization.

D. Dependence on Javassist

MiniLib utilizes certain functionalities of Javassist’s Application Programming Interface (API), as a result it is highly dependent on that. The use of alternative open-source libraries such as ASM (<https://asm.ow2.io/>) is considered, in order to explore potential performance gains (execution speed, memory consumption) or allow the use of extra features, such as the applicability to additional languages like Kotlin.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have present MiniLib, an approach that removes from the final executable any unused functionalities that may be present in the libraries, reducing attack surface and thus enhancing security. MiniLib identifies the used functionalities through static code analysis, ensuring efficiency, while both direct and indirect dependencies are considered, providing thus full coverage of the program model.

The MiniLib approach operates at the executable code level, warranting thus its usage for all applications. The efficiency of MiniLib has been validated through its application on applications drawn from the O-RAN 6G framework. These experiments have demonstrated a considerable reduction in the attack surface, which is reflected in the reduction of the application vulnerability overall score by a ratio ranging from 10.9% to 52.5%. Additionally, the application code size is reduced by a ratio ranging from 37% to 54%.

Future work will focus on extending the application functionality to use traces from dynamic program executions, in order to identify code fragments that are listed within methods that are activated during program execution but are never executed, e.g. due to the fact that associated conditions are never satisfied. This will enable the further removal of unused library methods that may entail vulnerabilities. Moreover, since the removal of methods from the final executable may render some class fields unreachable, in future versions of the MiniLib system we will consider the detection and removal of such fields. A more comprehensive evaluation using additional, non Spring-based applications is also planned, while the security reduction metrics will be further refined, considering remote exploitability of vulnerabilities and attack graphs.

The performance of the MiniLib system can be improved to promote its use in contexts where application deployment latency should be minimized. Towards this direction, we are exploring the use of concurrency, in order to process different execution branches in parallel, and therefore minimize the time

needed to produce the minified application.

Java, by default, exposes core methods such as `exec()`. This could have severe effects in the event that a threat actor exploits this method to execute malicious code. Similar methods, for example `delete()`, `renameTo()`, `listFiles()`, `list()` in `java.io.File` or the classes `FileInputStream`, `FileOutputStream`, `FileReader`, `FileWriter`, `RandomAccessFile`, should also be appropriately safeguarded if they are not removed by the minification software, to mitigate the risk.

Maven dependencies directory should be declared in order to parse JAR files into the class-pool. Virtual environments could be implemented in later implementations, where dependencies will be downloaded and handled outside of Maven's directories. Moving separation out of the application context will reduce MiniLib's workload for analyzing the libraries.

REFERENCES

- [1] W. Bangerth and T. Heister, "What makes computational open source software libraries successful?" *Computational Science and Discovery*, vol. 6, no. 1, p. 015010, Nov. 2013. [Online]. Available: <http://dx.doi.org/10.1088/1749-4699/6/1/015010>
- [2] J. Jenkov. (2019) Build a fat jar with maven. [Online]. Available: <https://jenkov.com/tutorials/maven/maven-build-fat-jar.html>
- [3] P. Ferreira, F. Caldeira, P. Martins, and M. Abbasi, *Log4j Vulnerability*. Springer International Publishing, 2023, p. 375–385. [Online]. Available: http://dx.doi.org/10.1007/978-3-031-33261-6_32
- [4] S. A. Haryono, H. J. Kang, A. Sharma, A. Sharma, A. Santosa, A. M. Yi, and D. Lo, "Automated identification of libraries from vulnerability data: can we do better?" in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ser. ICPC '22. ACM, May 2022, p. 178–189. [Online]. Available: <http://dx.doi.org/10.1145/3524610.3527893>
- [5] K. Kim, J. Kim, and S. Lee, "Stackguard+StackGuard+: Interoperable alternative to canary-based protection of stack smashing," *Electronics Letters*, vol. 60, no. 19, October 2024. [Online]. Available: <http://dx.doi.org/10.1049/ell2.13310>
- [6] B. M. Calatayud and L. Meany, "A comparative analysis of buffer overflow vulnerabilities in high-end iot devices," in *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, January 2022. [Online]. Available: <http://dx.doi.org/10.1109/CCWC54503.2022.9720884>
- [7] P. Sotiropoulos, C.-M. Mathas, C. Vassilakis, and N. Kolokotronis, "Minimizing Software-Rooted Risk through Library Implementation Selection," in *Proceedings of the 2023 IEEE CSR Workshop on Data Science for Cyber Security (DS4CS)*, July 2023.
- [8] I. Sayar, A. Bartel, E. Bodden, and Y. Le Traon, "An in-depth study of java deserialization remote-code execution exploits and vulnerabilities," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, Feb. 2023. [Online]. Available: <https://doi.org/10.1145/3554732>
- [9] Oracle. (2023) Uses of class `java.lang.ProcessBuilder`. [Online]. Available: <https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/class-use/ProcessBuilder.html>
- [10] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 941–955.
- [11] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 340–353. [Online]. Available: <https://doi.org/10.1145/1102120.1102165>
- [12] M. Alhanahnah, Y. Boshmaf, and A. Gehani, "Sok: Software debloating landscape and future directions," in *Proceedings of the 2024 Workshop on Forming an Ecosystem Around Software Transformation*, ser. FEAST '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 11–18. [Online]. Available: <https://doi.org/10.1145/3689937.3695792>
- [13] H. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications," in *Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research, FoSER 2010*, 11 2010, pp. 421–426.
- [14] C. Qian, H. Hu, M. Alharthi, S. P. H. Chung, T. Kim, and W. Lee, "Razor: A framework for post-deployment software debloating," in *USENIX Security Symposium*, 2019. [Online]. Available: <https://api.semanticscholar.org/CorpusID:198978919>
- [15] S. Bhattacharya, M. G. Nanda, K. Gopinath, and M. Gupta, "Reuse, recycle to de-bloat software," in *Proceedings of the 25th European Conference on Object-Oriented Programming*, vol. 6813, 07 2011, pp. 408–432.
- [16] C. Porter. (2023) Compiler and machine learning-based predictive techniques for security enhancement through software debloating. [Online]. Available: <https://hdl.handle.net/1853/75575>
- [17] I. Agadakov, D. Jin, D. Williams-King, V. Kemerlis, and G. Portokalidis, "Nibbler: debloating binary shared libraries," in *ACSAC '19: Proceedings of the 35th Annual Computer Security Applications Conference*, 12 2019, pp. 70–83.
- [18] I. Agadakov, N. Demarinis, D. Jin, K. Williams-King, J. Alfajardo, B. Shteinfeld, D. Williams-King, V. Kemerlis, and G. Portokalidis, "Large-scale debloating of binary shared libraries," *Digital Threats: Research and Practice*, vol. 1, pp. 1–28, 12 2020.
- [19] J. He, P. Hou, J. Yu, J. Qi, Y. Sun, L. Li, R. Zhao, and Y. Wu, "D-linker: Debloating shared libraries by relinking from object files," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, pp. 3768–3779, 11 2024.
- [20] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo, "Knowledge-based library re-factoring for an open source project," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, 02 2002, pp. 319 – 328.
- [21] G. Antoniol and M. Di Penta, "Library miniaturization using static and dynamic information," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, 10 2003, pp. 235 – 244.
- [22] B. Bruce, T. Zhang, J. Arora, H. Xu, and M. Kim, "Jshrink: in-depth investigation into debloating modern java applications," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 11 2020, pp. 135–146.
- [23] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 10 2018, pp. 380–394.
- [24] A. Quach, A. Prakash, and L. Yan, "Debloating software through Piece-Wise compilation and loading," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 869–886. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity18/presentation/quach>
- [25] C. Soto-Valero, T. Durieux, N. Harrand, and B. Baudry, "Coverage-based debloating for java bytecode," *ACM Transactions on Software Engineering and Methodology*, vol. 32, 07 2022.
- [26] H. Koo, S. Ghavamnia, and M. Polychronakis, "Configuration-driven software debloating," in *Proceedings of the 12th European Workshop on Systems Security*, 03 2019, pp. 1–6.
- [27] M. Panahi, T. Harmon, and R. Klefstad, "Adaptive techniques for minimizing middleware memory footprint for distributed, real-time, embedded systems," in *2002 14th International Conference on Ion Implantation Technology Proceedings (IEEE Cat. No.02EX505)*, 11 2003, pp. 54 – 58.
- [28] Q. Xin, Q. Zhang, and A. Orso, "Studying and understanding the tradeoffs between generality and reduction in software debloating," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 01 2023, pp. 1–13.
- [29] S. Nadschläger, D. Hofer, J. Küng, and M. Jäger, "Data structures for a generic software system using the composite design pattern," in *26th European Conference on Pattern Languages of Programs*, ser. EuroPLoP'21. ACM, Jul. 2021, p. 1–7. [Online]. Available: <http://dx.doi.org/10.1145/3489449.3489972>
- [30] V. Sarcar, *Visitor Pattern*. Apress, 2022, p. 513–547. [Online]. Available: http://dx.doi.org/10.1007/978-1-4842-7971-7_24
- [31] N. Vermeir, *Runtimes and Desktop Packs*. Apress, 2022, p. 21–29. [Online]. Available: http://dx.doi.org/10.1007/978-1-4842-7319-7_2