electronics

Article

A Software Vulnerability Management Framework for the Minimization of System Attack Surface and Risk

Panagiotis Sotiropoulos †, Christos-Minas Mathas †, Costas Vassilakis *,† and Nicholas Kolokotronis †

Department of Informatics and Telecommunications, University of the Peloponnese, 221 31 Tripoli, Greece; panossot@uop.gr (P.S.); mathas.ch.m@uop.gr (C.-M.M.); nkolok@uop.gr (N.K.)
* Correspondence: costas@uop.gr; Tel.: +30-2710-372203
† These authors contributed equally to this work.

Abstract: Current Internet of Things (IoT) systems comprise multiple software systems that are deployed to provide users with the required functionalities. System architects create system blueprints and draw specifications for the software artefacts that are needed; subsequently, either custom-made software is developed according to these specifications and/or ready-made COTS/open source software may be identified and customized to realize the overall system goals. All deployed software however may entail vulnerabilities, either due to insecure coding practices or owing to misconfigurations and unexpected interactions. Moreover, software artefacts may implement a much broader set of functionalities than may be strictly necessary for the system at hand, in order to serve a wider range of needs, and failure to appropriately configure the deployed software to include only the required modules results in the further increase of the system attack surface and the associated risk. In this paper, we present a software vulnerability management framework which facilitates (a) the configuration of software to include only the necessary features, (b) the execution of security-related tests and the compilation of platform-wide software vulnerability lists, and (c) the prioritization of vulnerability addressing, considering the impact of each vulnerability, the associated technical debt for its remediation, and the available security budget. The proposed framework can be used as an aid in IoT platform implementation by software architects, developers, and security experts.

Keywords: IoT systems; software vulnerabilities; risk management; technical debt; system design; system security

Citation: Sotiropoulos, P.; Mathas, C.-M.; Vassilakis, C.; Kolokotronis, N. A Software Vulnerability Management Framework for the Minimization of System Attack Surface and Risk. *Electronics* **2023**, *12*, 2278. https://doi.org/10.3390/electronics12102278

Academic Editor: Seokjoo Shin

Received: 10 March 2023
Revised: 2 May 2023
Accepted: 15 May 2023
Published: 18 May 2023

1. Introduction

The Internet of Things (IoT) concept involves devices with Internet connectivity, which can exchange data, information, and services, obtain data from their environment through sensors, and initiate changes to it through actuators. The building blocks for IoT systems exhibit considerable diversity, ranging from specialized industrial or enterprise products, such as production line robots [1,2], smart grid devices including smart meters [3], connected and autonomous cars [4,5], or consumer products including wrist bands and smart watches, smart air conditioners, and smart TVs. IoT systems are expected to proliferate over the next years: Statista projects that the number of connected IoT devices will double from 2023 to 2030 [6], while IHS Markit predicts that the number of connected IoT devices will exhibit an annual increase of 12%, escalating to 125 billion devices in 2030 [7].

A critical factor for the operation of existing IoT systems and the proliferation of new ones is security. Multiple studies e.g., refs. [8–11] have identified a number of key security areas, spanning across physical aspects, the network layer, the edge layer and the application layer. These include physical device protection, architectural concerns, authentication, encryption, trust, secure routing protocols, privacy concerns, and so forth. A major factor affecting the security of IoT systems, at virtually any layer, is the software. All deployed software (either custom-made to serve the requirements of the particular system,

or ready-made "Commercial off-the-Shelf (COTS)"/open-source software that has potentially been customized) may entail vulnerabilities, which may be exploited by attackers. Characteristically, the study reported in [12] reports that after security tests were conducted on thirteen routers and network attached storage (NAS) devices for small office/home office (SOHO) environments, a number of vulnerabilities was discovered in each of them, totalling 125 common vulnerabilities and exposures (CVEs). Ref. [12] also reports that manufacturers were notified about the vulnerabilities, however the number of responses collected was very limited, and very few of the vulnerabilities were actually addressed.

The increased number of vulnerabilities in IoT devices, coupled with the large number of these devices and their often unrestricted accessibility through the internet, has led to some world-scale security incidents affecting millions of devices, including Hajime [13], BASHLITE [14], and Log4Shell [15].

Software-rooted vulnerabilities occur either due to insecure coding practices due to misconfigurations and unexpected interactions (e.g., race conditions). Moreover, deployed software artefacts may be poorly customized, and offer a broader set of functionalities than may be strictly necessary for the system at hand: for instance, if the OpenLiberty server [16] is deployed at some system, the Java Message Service 2.0 [17] feature may be left at an enabled state, although it is not needed in the particular installation. In such a case, the attack surface of the system increases (all APIs/endpoints of the Java Message Service 2.0 can be used by attackers), and any vulnerabilities present in the code realising the Java Message Service can be targeted for exploitation.

Addressing software-rooted vulnerabilities in an IoT system is thus a complex, multi-faceted issue, involving (a) the minimization of the vulnerabilities present in the code, through the inclusion of only the necessary features, (b) the identification of the remaining vulnerabilities, and (c) the fixing of these vulnerabilities. However, fixing the vulnerabilities incurs a cost—in both time and human resources—and the time available or the budget allocated to this task may be limited, not permitting the tackling of all issues. In such cases, the effort of the developer team should be directed to fixing the errors that would minimize the overall *residual risk*, i.e., the risk owing to the vulnerabilities that will not be fixed, due to security budget constrains.

To the best of our knowledge, no commercial system or research proposal offers prioritization of software components' vulnerability addressing the associated technical debt for its remediation and the available security budget (considering the impact of each vulnerability); developers are not adequately supported to prioritize fixes and deploy platforms with minimized residual risk.In this paper, we present a software vulnerability management framework which supports all the stages of a pipeline for the management of IoT platform software vulnerabilities, i.e., (a) the configuration of software to include only the necessary features (b) the execution of security-related tests and the compilation of platform-wide software vulnerability lists, (c) the estimation of the impact and the associated fixing cost for each vulnerability, and (d) the prioritization of vulnerability addressing the associated technical debt for its remediation and the available security budget, considering the impact of each vulnerability. The proposed framework can be used as an aid in IoT platform implementation, by software architects, developers, and security experts.

The work presented in this paper advances the state-of-the-art by (a) proposing a statistics-based method for the estimation of impact of detected vulnerabilities, (b) proposing an integer programming-based algorithm for prioritizing security fixes with the goal of minimizing the residual risk level, and (c) harnessing the power of a test management framework [18] and static code analysis [19], and combining them with the estimation of impact of detected vulnerabilities and the integer programming-based prioritization algorithm to synthesize a comprehensive framework for the security analysis of platform software which formulates proposals on the prioritization of addressing security issues, taking into account the software components to be included and their features, the impact of

each vulnerability within the source code, the associated technical debt for its remediation, and the overall available security budget.

In the remainder of this section we initially present the landscape of software-rooted security issues in IoT systems, as reported in papers where the authors exploited (a) security testing tools, (b) lists of known vulnerabilities, and (c) catalogues of common weaknesses found in IoT software (Section 1.1). Note that these works aim to identify the issues that may be present in the software without providing tools for assessing their impact on specific configurations or platform deployments and without providing any means for prioritizing fixing issues. Subsequently, we present the state-of-the-art in the security assessment of IoT software code (Section 1.2). In this subsection, we focus on static code analysis techniques [20], which are employed by the proposed framework. We also briefly cover other techniques, including dynamic testing and fuzzing, to provide a more complete picture of the software security assessment methods.

In Section 2 we present the proposed method for the management of vulnerabilities in IoT platform software, while in Section 3 we demonstrate the application of the framework in IoT platform software. Finally, in Section 4 the practical and theoretical implications of the proposed framework are discussed, and future work is outlined.

1.1. Software-Related Security Issues in IoT Software

According to OWASP [21], a large percentage of the vulnerabilities present in applications can be linked to insecure coding practices followed by software developers. Even though extensive research has been published on the analysis of IoT vulnerabilities, it is focused mainly on black-box methods such as penetration testing and fuzzing [sachidanada][samtani][geneiatakis][overstreet]. There is limited research available on the white-box testing of open-source software for IoT devices, which involves examining the source code to identify vulnerabilities. Source code reviews could be significantly beneficial for vulnerability research in the IoT field, as they often detect different vulnerabilities than those revealed through black-box techniques. Additionally, the detailed documentation of vulnerabilities in open-source software constitutes useful information for preventing similar coding mistakes from occurring in the future [22].

In Schiller et al. [23], the authors discuss the landscape of IoT security by first providing some background information on IoT in general, on the concept of security, IoT networking, and the available IoT architectures. Furthermore, they discuss challenges in achieving IoT security and propose a threat taxonomy. The authors composed the taxonomy by reviewing the available research and by integrating the threats and attack methods identified. The classification system used organizes the threats according to a three-level architecture model of the IoT which includes the sensing, network, and application layers. In the threats against IoT classified under the application layer, we can find several that can be traced back to insecure coding practices, some on a higher level and some on a lower level: data modification, elevation of privilege, DoS, password change, password guessing, buffer overflow, memory corruption, code execution, SQL injection, XSS, and CSRF.

In Calatayud et al. [24], the authors utilize the Raspberry Pi hardware platform as a base to test operating systems used in high-end IoT devices against a multitude of buffer overflow attack forms. The attacks were carried out using the RIPE tool [wilander]modified accordingly for the ARM architecture. The operating systems were tested with and without buffer overflow protections in place. The outcomes offer valuable information about common buffer overflow vulnerabilities found in IoT operating systems along with a persistent pattern of the preventive measures that are used to protect against this type of attack. The target operating systems are RPi OS, Xubuntu, Alpine Linux, Arch Linux ARM, and Chromium OS. RPi OS v10 was found vulnerable to 12 attacks with the vulnerable functions being memcpy and homebrew. Xubuntu v18.04 and Alpine Linux v3.14 were found to be vulnerable to 50 attacks with the vulnerable functions being memcpy, strcpy, sprint, and fscan. Arch Linux ARM v5.10 was vulnerable to 10 attacks with vulnerable functions being

the same ones as with RPi OS. Lastly, in Chromium OS v5.4, 59 attacks were successful with the set of vulnerable functions being the same as with Xubuntu and Alpine Linux.

In Al-Boghdady et al. [19], the authors investigate the security posture and the vulnerabilities present in the source code of four popular IoT operating systems through static analysis. They utilize three static analysis tools—Cppcheck, Flawfinder, and RATS—to analyze sixteen different versions of the four C/C++ IoT OSs with the research goals being the identification of vulnerabilities from the common weakness enumeration (CWE) scheme and to find out if the security errors and their density (errors per 1K source lines of code) increase or decrease over time. Furthermore, the last research question posed by the authors asks what the relationship between the vulnerabilities of the IoT OSs and their evolutionary properties is. The tool CodeScene is utilized to that end. The results showed that while the total number of security errors increases with each version, the error density decreases over time for all examined operating systems, with few exceptions. The most prevalent vulnerabilities in the OSs examined for Cppcheck were CWE-561 (dead code), CWE-398 (7PK—code quality), and CWE-563 (assignment to variable without use), while CWE-119 (improper restriction of operations within the bounds of a memory buffer), CWE-120 (buffer copy without checking zize of input ("classic buffer overflow')), and CWE-126 (buffer over-read) for Flawfinder. For RATS, it was CWE-119, CWE-120, and CWE-134 (use of externally-controlled format string).

Mathas et al. [22] evaluate the vulnerabilities in IoT software used mainly in smart grid applications through static analysis of the source code. The authors have analyzed open-source software which could be used at any level of the software stack, including operating system level, application level, and library level, in order to obtain a comprehensive understanding of the relevant vulnerability landscape. The assessed software includes jSML, lib60870, libiec61850, JavaSMQ, Pymodbus, Modbus4j, Minnow server, Boa Webserver, thttpd, MicroWebSrv 2, and Busybox. The static application security testing (SAST) is performed by utilizing the SonarCloud (https://sonarcloud.io, accessed on 14 May 2023) and Codacy (https://app.codacy.com, accessed on 14 May 2023) platforms.

The results of the two platforms are manually reviewed to discern between true and false positives. The final results are categorized based on a customized vulnerability categorization scheme which was created by combining the OWASP Top 10 list and the MITRE common weakness enumeration (CWE) scheme. Based on the results received from the static analysis, the custom categorization scheme includes improper certificate validation, buffer overflow, weak cryptography, sensitive data exposure, race condition, and broken access Control. Furthermore, the frequency and the potential impact of the identified vulnerabilities is considered. The article provides a detailed examination of true and false positives which can assist both researchers and practitioners to better focus on the areas requiring review. The vulnerabilities results report 2 for the improper certificate validation category, 6 for buffer overflow, 23 for weak cryptography, 59 for sensitive data exposure, 8 for race condition, and 3 for broken access control. Additionally, the results show numerous false positives reported by the SAST tools utilized.

1.2. Assessing the Security of IoT Software

Static code analysis [20] is a part of the security development lifecycle [25] and is performed with the code review of static code, which can also be realized running static code analysis tools to detect possible security vulnerabilities by analyzing the code of the tested software. Static analysis can be used with both source and compiled code. While the development flaws detected through static analysis may include non-security-related issues as well, in this work, we focus only on security issues. Static analysis with a focus on security issues is usually referred to as static application security testing (SAST). SAST identifies problematic patterns by checking the code statically rather than inspecting it during runtime. Depending on the implementation, SAST can be simple or complex, and can detect patterns in the code, produce control graphs, or analyze data flow logic to identify user input that reaches sensitive code segments [26].

SAST has become an easy and efficient practice that has gained widespread acceptance in recent years. It is available in various forms, including IDE plugins, standalone applications, online services, and solutions integrated into continuous integration/continuous delivery (CI/CD) pipelines [27]. SAST tools are available in both open-source and commercial formats, offering different functionalities that cater to specific needs. Various techniques are employed by static analyzer tools, such as data flow analysis, control flow graphs (CFGs), taint analysis, and lexical analysis [28]. However, even with recent advancements in SAST, tools still have high false-positive rates, necessitating human intervention for results evaluation [26]. One basic limitation of SAST solutions is that they suffer from a high rate of false positives in their results [22]. To that end, we begin this section by presenting one of the latest works towards improving SAST for IoT by utilizing machine learning. SAST is an invaluable way of assessing the security of IoT but not the only one needed as it is meant to be complementary (but necessary) to the more traditional black-box techniques used. Black-box methods used for vulnerability detection in IoT systems include, among others, fuzzing, taint-analysis, symbolic execution, homology analysis, and penetration testing [29–33]. In the remainder of this section, we discuss some of the latest scientific work conducted on these methods.

Kotenko et al. [34] propose an intelligent framework concept for the static analysis of IoT systems utilizing machine learning techniques. They systematize the fundamental components of static analysis and machine learning areas to form two models: the SA (static analysis) model and the ML (machine learning) application model for SA. SA is broken down into stages and ML into tasks. The models are represented as matrices with rows corresponding to the tasks and columns to the stages.

The SA stages considered are data collection, data preparation, data processing, and result formation (the columns of the SA model matrix). For the assignment of each activity to the stages of SA, they utilize the following formalization: any given data is represented in terms of its content (C), which refers to the information it contains, and its form (F), which pertains to its appearance. Thus, data stored within the IoT system and modified during the SA process can be expressed as a tuple $<F \mid C>$. Based on this, a formalized description of each stage is defined.

The ML tasks were chosen based on both the general theory and a large number of scientific papers and their reviews. The ML tasks considered are classification, anomaly detection, regression, clustering, and generalization (the rows of the SA model matrix). The tasks are described through formalized definitions as well. The resulting matrix contains in its cells a formalized record of stage actions based on the solution of one of the ML tasks. The authors note that non-ML statements will precede and succeed the ML ones, since apart from the intelligent component, each step comprises strictly defined rules (e.g., unpacking archives with files, ranking documents by their size, etc.). The matrix's validity was confirmed through expert analysis.

For the second part, an analysis of research papers relevant to analyzing IoT systems is conducted. Each study is categorized and assigned based on its attributes to one or more SA stages and one or more ML tasks. The second matrix has the same types of rows and columns, but the cells are the research papers that were categorized under the corresponding stage-task couple.

The two models make it possible to create methodological solutions that are theoretically and practically sound in order to provide information security in the IoT systems domain. This, of course, necessitates the development of a suitable framework that can ensure the execution of all phases utilizing the wide range of ML methods available for big data and heterogeneous data. The novelty of this work resides in the fact that it takes into consideration the phases of data collection and preparation that precede the code analysis. Thus, this work defers from previous ones in that it covers the entirety of the SA process. Additionally, this review is the first to consider the full range of ML solutions, both theoretically and practically, for each stage of SA. Finally, not only is SA divided into stages, but it is also suggested to represent the actions of these stages in a formalized

manner, which involves transforming the form and content of the data being studied in the IoT system.

He et al. [33] propose a homology detection method based on a clonal selection algorithm for detecting vulnerabilities in IoT firmwares. The proposed problem indicates that methods utilizing machine learning for detecting vulnerabilities in IoT firmwares need sample data of firmware vulnerabilities. This sample data is very scarce for some types of vulnerabilities. The authors characterize the results achieved by machine learning algorithms on this matter as "not ideal" and attribute this to their need for a large set of sample data. To that end, this work proposes a firmware vulnerability homology detection method based on the clonal selection algorithm combined with the simulated annealing algorithm. Unlike preexisting machine learning methods, this method relies solely on the affinity between the objective function and the detector, eliminating the need for extensive sample data sets. In the clonal selection algorithm of biology, the immune system identifies antigens and creates a variety of plasma cells to produce antibodies that are customized based on the specific characteristics of each antigen. Antibodies that have a high affinity to their respective antigens are retained, while those with low affinities are discarded. When applied to the method in question, the optimization problem is the antigen, the feasible solution to the optimization problem is the antibody, and the quality of the feasible solution is represented by affinity. Essentially, the antigen is a vulnerability function, and affinity represents the similarity between the sample function and the original one. The simulated annealing algorithm is a general optimization algorithm. It begins from a high initial temperature and randomly explores the solution space to find the global optimal solution of the objective function. The main idea is to use the probability of a local optimal solution for obtaining the global optimal solution. This work combines the efficient local search capability of simulated annealing with the rapid convergence of the clonal selection algorithm. The proposed method is tested against a neural network algorithm that attempts to tackle the same issue. The evaluating indicators are recall rate and accuracy. The recall rate is the ratio of the number of actual vulnerabilities in the algorithm's vulnerability set and the total number of vulnerabilities that should be detected. The accuracy indicator is the ratio of the number of actual vulnerabilities in the algorithm's vulnerability set and the total number of vulnerabilities predicted by the algorithm. The proposed method was found to be faster than its counterpart, as well as having a significantly higher recall rate, improving it by about 13%. There is a slight improvement in accuracy as well.

In Akhilesh et al. [35], the authors describe an automated penetration testing framework for smart home IoT devices. The goal and novelty of the work lies in the full automation of the framework and the ease of use by both technical and non-technical users so that anyone can assess the security of the devices deployed in her/his smart home network. The authors begin by reviewing and comparing the relevant research discussing differences and limitations in order to choose the most suitable work to adopt. The method selected [luis costa] is based on the penetration testing execution standard (PTES). The framework is designed to detect the five most common vulnerabilities for smart home IoT devices. The five most common vulnerabilities were chosen by the authors based on the OWASP IoT Top 10, the work they adopted [luis costa], and the OWASP Top 10. The resulting list includes insecure web interface, remote access vulnerability (improper authentication), insecure network services, lack of transport encryption, and insecure Firmware/Software. The framework can be divided into five consecutive parts: reconnaissance, check for remote access vulnerability, check for insecure web interface, automated traffic capture, and vulnerability detection through traffic analysis. The framework is written as a Python program which executes a combination of tools in a specific order at the user's discretion. Each vulnerability is assigned a corresponding method and one or more tools used to detect it. The tools utilized by the framework are Net Discover, Nmap, OWASP ZAP (zap-cli), Medusa, WhatWeb, Wireshark (t-shark and pyshark), Binwalk, and Firmwalker. The framework was executed in a home network with the following devices connected to it: Tp-Link SmartPlug, Tp-Link Smart bulb, Tp-Link Smart Camera, Google Home Mini, and LIFX Smart Bulb. For

the Tp-Link SmartPlug, a probable network services vulnerability was reported. Lack of transport encryption and insecure firmware vulnerabilities were reported for the Tp-Link Smart Bulb and Smart Camera, while no vulnerabilities were reported for Google Home Mini and LIFX Smart Bulb. The execution time varied from 1 to 8 s for each device scan. Additionally, the authors calculated the CVSS scores of the detected vulnerabilities. The base scores were summarized to form a total score for each device. The Tp-Link Smart Bulb and Smart Camera were found to be the most vulnerable. Google Home Mini had the same score as LIFX Smart Bulb (zero), but further analysis conducted by the authors showed that Google Home Mini employs more secure mechanisms than its counterpart and is therefore the most secure.

Zheng et al. [36] propose a novel approach to greybox fuzzing for Linux-based IoT devices. Greybox fuzzing is a very effective vulnerability discovery technique but when applied to IoT devices it faces various limitations. The basic limitation occurs from the IoT application's high reliance on specific system environments and hardware. Some techniques use full-system emulation to bypass that limitation, but this technique has a high overhead. To that end, some works, such as Firm-AFL, propose to combine full-system emulation with user-mode emulation in order to provide full compatibility in user emulation. Firm-AFL executes the application in user-emulation mode until a system call is needed to continue the execution. When that occurs, the emulation shifts to full-system emulation to execute the system call. This approach proved to be less efficient than full-system emulation, especially in the cases of applications that make frequent system calls. To that end, the authors propose EQUAFL, a greybox fuzzing framework with enhanced user-mode emulation. EQUAFL first executes the application in a full-system emulation and observes the key points, such as the setting of launch variables, the generation of configuration files, and network setup, etc. Next, EQUAFL sets up the execution environment for the application by replaying the observed behaviors. This called an observe–replay strategy by the authors. The framework's performance is evaluated on compatibility, efficiency, and vulnerability discovery. Two different datasets were used as benchmarks. The first dataset comprises two standard benchmarks and the other dataset comprises 70 embedded firmware images from D-Link, TRENDnet, and NETGEAR. The first dataset is small and does not contain bugs, so it is used for the evaluation of efficiency and compatibility. The second dataset is used for vulnerability discovery as well. The baselines used are simple user-mode emulation, full-system emulation, and Firm-AFL. Additionally, experiments are conducted 5 times each to mitigate the randomness of the fuzzers. The compatibility results show the successful execution of the first dataset and 66 out of 70 applications in the second (real world) dataset. EQUAFL's results are much better than simple user emulation and comparable to full-system emulation in terms of compatibility. In terms of efficiency EQUAFL was found to be 26 times faster than full-system emulation and 14 times faster than Firm-AFL in real-world applications. The overhead of EQUAFL on the benchmark data was marginal compared to user-mode emulation. It discovered 10 vulnerabilities, for 6 of which CVEs were assigned after reporting them to the corresponding vendors. Finally, EQUAFL was shown to detect vulnerabilities much faster than its counterparts.

2. Materials and Methods

In this section we present the proposed framework. Firstly, in Section 2.1 we describe the overall framework architecture, while in the remaining subsections we provide details on the operation of each architectural component.

2.1. Architecture of the Proposed Framework

The overall architecture of the proposed framework is illustrated in Figure 1. The processing pipeline begins by gathering the software components that will be deployed on the IoT platform. The source of each such software component is stored in the corresponding code repository. This step is instrumented by the additional testsuite framework [18] and detailed in Section 2.2.

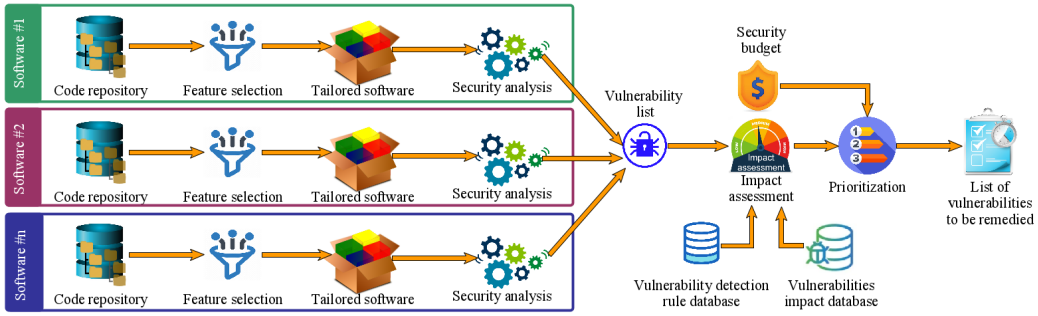

Figure 1. Proposed framework architecture.

Subsequently, feature selection is applied to each software component, producing the respective tailored software. Recall from the introduction that feature selection is a step that assists in limiting the attack surface and the associated risk, while it can also lead to configurations with smaller memory footprint and resource requirements. The outcome of the feature-selection step is a set of software components that are tailored to the needs of the specific IoT platform deployment. In order to perform feature-based tailoring of software components, the proposed framework utilizes the relevant capabilities of the additional testsuite framework [18]; the additional testsuite framework is summarized in Section 2.2, while Section 2.3 provides details on the feature selection capabilities of the additional testsuite framework.

Afterwards, each software component is analyzed to identify security issues, forming a list of software-related vulnerabilities related to the individual software component. In this context, the additional testsuite framework submits the relevant projects to the SonarQube static code-analysis platform, and for each project, it gathers the results through the SonarQube API https://sonarqube.inria.fr/sonarqube/web_api/api/ (accessed on 14 May 2023). Project-specific lists are then merged to formulate a comprehensive vulnerability list for the whole of the platform. Security analysis is presented in Section 1.2.

Each vulnerability in the comprehensive platform-wide list is assessed to estimate its impact on the platform. To this end, a statistical approach is used. Vulnerability impact estimation is discussed in Section 2.5.

Finally, the vulnerabilities are prioritized, taking into account the impact of each vulnerability and the cost to fix it, as well as the overall security budget, producing the final list of vulnerabilities to be remedied. This is accomplished through an integer programming-based optimization scheme, which is detailed in Section 2.6.

This list can guide developers to the process of maintaining the software so as to minimize the overall *residual risk*, i.e., the risk owing to the vulnerabilities that will not be fixed, due to security budget constrains.

2.2. The Additional Testsuite Framework: An Overview

The additional testsuite framework (ATF) [18] is novel approach for the management of code testsuites, providing relevant structures and instrumentation. ATF supports a multitude of features, including management of tests for multi-version applications, test-driven development, dynamic/selective program builds, feature-based builds, testing in different environments, and source code analysis. ATF utilizes annotations to associate tests with specific application characteristics, and dynamically matches these characteristics against build specifications and/or deployment environment attributes to (a) retrieve from the source code repositories the relevant sources to be used for the specific build, (b) create the executable image of the tailored software component according to the build specifications, and (c) deploy the executable image to the designated deployment environment.

The benefits stemming from the introduction and use of ATF include the following:

1. Tests are written once, and can be flexibly associated with any number of software programs and versions, limiting the effort and complexity needed for the maintenance of test cases.
2. It supports dynamic/selective program builds that include only the portions of the software that match some designated functionality.
3. For software applications that are developed or organized according to the featured-based development paradigm [37], builds can be tailored to create executables that only support a subset of the available features.
4. It can underpin the localization of bugs introduced during software evolution, including regression bugs, through the comparison of code in versions producing erroneous results against the code in versions yielding correct results.
5. It can facilitate documentation compilation, since functionality-oriented and feature-based tests can be included in documentation on the functionality/feature they pertain to, serving as examples of the specific functionality/feature as well as providing examples of usage.

For more information regarding the capabilities and functionality of the ATF, the interested reader is referred to [18]. In the next subsection, we describe the feature management functionality of ATF, which are utilized in the context of the proposed software vulnerability management framework.

2.3. Feature Management Using the ATF

A software program can be defined as the unit of code parts that implement a set of features that are intended as the provided functionalities of the software [37]. Each feature of a software system is an optional or incremental unit of functionality, and is associated with relevant code that realizes this functionality [38,39]. While software programs constantly evolve to accommodate an increasing number of features, specific deployments of these programs in the context of IoT platforms may necessitate and utilize only a limited number of the available features. Configuring the software program so as to only include the code that realizes the actually needed features may contribute to decreasing the attack surface and minimizing the memory footprint.

ATF provides support for the testing of programs that are dynamically tailored to specific needs through the usage of the `FEATURE_LIST` tailoring specification and the `@ATFeature` code annotation. In more detail, when execution tailoring commences:

1. ATF consults the environment variable `FEATURE_LIST`, which includes the path to a feature-tailoring configuration file listing the features that should be enabled in the specific software build; for each feature, the relevant version that should be enabled is also specified, as illustrated in Listing 1. Then, ATF arranges so that the tailored software bundle includes the relevant code realizing the specific features, retrieving the respective code from relevant repositories.
2. ATF scans the code for instances of the `@ATFeature` annotation; this annotation is associated to methods and specifies the features that need to be enabled for the method to be included in the final executable, effectively thus providing an advanced conditional compilation mechanism. More specifically, the `@ATFeature` annotation lists the program features that the specific method is dependent on, and during the tailoring procedure the ATF matches these features against the feature tailoring configuration specified via the `FEATURE_LIST` environment variable, and arranges so that the method implementation code is included in the tailored version of the software if all the specified features are enabled via `FEATURE_LIST` and the version of each enabled feature also matches the designated version range. Listing 2 presents an example of the usage of the `@ATFeature` annotation.

Listing 1. Example `FEATURE_LIST` file contents.

```
jaxrs,2.3.1
jaxb,2.4.0
jsonp,1.2
cdi,2.1
localConnector,1.1
servlet,4.1.34
```

Listing 2. Example usage of the `@ATFeature`.

```
@ATFeature(feature ={"jaxrs, jaxb, jsonp, cdi, localConnector, servlet"},
minVersion ={"2.1, 2.2, 1.1 ,2.0, 1.0, 4.0"},
maxVersion ={"null, null, null, null, null, null"})
public void doJaxRs () throws Exception {// Feature-dependent code}
```

2.4. Static Code Analysis for Vulnerability Detection

In the context of the proposed software vulnerability management framework, the facilities of the additional testuite framework are used to gather the tailored software components and submit them to static code analyzers. In our current configuration, the SonarQube static code analyzer is employed; in particular, the SonarQube API (https://sonarqube.inria.fr/sonarqube/web_api/api, accessed on 14 May 2023) is used to submit tailored software components for analysis and retrieve the analysis results, which are filtered to contain only vulnerabilities, by setting the `types` REST API parameter to the value `VULNERABILITY`.

The SonarQube analyzer returns for each security issue identified numerous information items which include:

- A textual description of the issue;
- A designation of the estimated severity of the issue, which may be *INFO*, *MINOR*, *MAJOR*, *CRITICAL*, or *BLOCKER*;
- The component (directly identifying the source file) and the range of the code lines where the security issue was found;
- the SonarQube security rule that triggered the vulnerability flagging;
- An estimate of the technical debt associated with the vulnerability, i.e., the time needed to modify the code in order to eliminate the security issue.

2.5. Vulnerability Impact Estimation

Once the vulnerability management framework has determined the list of vulnerabilities present in the software, the next step is to estimate the impact that each of these vulnerabilities will have on the IoT platform. Recall (from Section 2.4) that the list of vulnerabilities contains, for each vulnerability, a reference to the SonarQube security rule that triggered the vulnerability flagging. This information is exploited by the vulnerability management framework to compute an estimate of the vulnerability impact, according to the following process:

1. The rule is looked up in the SonarQube rules database (https://rules.sonarsource.com/, accessed on 14 May 2023) and its full record is retrieved. This record includes:
    - *Common weakness enumeration (CWE) identifiers.* CWE identifiers are codes assigned to typical security-related code anti-patterns, i.e., patterns of code that are known to lead to vulnerabilities. For instance the *java/RSPEC-6437* SonarQube security rule (https://rules.sonarsource.com/java/RSPEC-6437, accessed on 14 May 2023) is linked to the *CWE-798—use of hard-coded credentials* (https://cwe.mitre.org/data/definitions/798.html, accessed on 14 May 2023) and the *CWE-259—of hard-coded password weaknesses* (https://cwe.mitre.org/data/definitions/259.html, accessed on 14 May 2023). These identifiers are saved and used in the vulnerability impact estimation, as described below.

- *A detailed description of the vulnerability*, including an explanation of the mechanics of the code anti-pattern, a substantiation of why the anti-pattern leads to vulnerabilities, and recommendations on how the code can be transformed to eliminate the vulnerability. This information is saved, to be presented to software security experts and to assist them in their vulnerability remediation tasks.

2. Subsequently, the vulnerability management framework applies a statistical approach to compute an estimate of the security issue. More specifically, the vulnerability management framework utilizes the information present in the common vulnerability enumeration (CVE) database (https://cve.mitre.org/data/downloads/, accessed on 14 May 2023) to identify known vulnerabilities that owed to the exact same code anti-patterns to which the current security issue is associated to. This database will be denoted as $VulDB$. For each vulnerability $vul \in VulDB$, the following fields are retrieved:
   - $id(vul)$, which corresponds to the id of the vulnerability
   - $weak(vul)$, which denotes the set of common weaknesses to which the vulnerability is associated. For instance, for the vulnerability with an ID equal to *CVE-2009-0003* it holds that $weak(CVE\text{-}2009\text{-}0003) = CWE\text{-}119$, i.e., vulnerability *CVE-2009-0003* is associated with the CWE having an ID equal to *CWE-119*, corresponding to the anti-pattern of improper restriction of operations within the bounds of a memory buffer (https://cwe.mitre.org/data/definitions/119.html, accessed on 14 May 2023), commonly referred to as *buffer overflow*.
   - $impact(vul)$, which corresponds to the impact of the vulnerability, i.e., a measure of the adverse effects that the exploitation of the vulnerability by attackers may have on the platform. The value of $impact(vul)$ is assigned by human experts, after careful review of the application code.

If the current security issue S_i is associated to weaknesses $W(S_i) = CWE_1, CWE_2, \ldots CWE_n$, then the impact estimate of S_i, which will be denoted as $IE(S_i)$, is computed, as shown in Equation (1):

$$IE(S_i) = \max_{cwe \in W(S_i)} \frac{1}{|Vuls(cwe)|} * \sum_{vul \in Vuls(cwe)} impact(vul) \quad (1)$$

where $Vuls(cwe) = v \in VulDB : cwe \in weak(v)$, i.e., the set of all vulnerabilities in $VulDB$ that are rooted to the particular CWE. Effectively, for each weakness that is associated with the current security issue, the average impact values of all known and human expert-assessed vulnerabilities rooted to the particular weakness are calculated, and finally the maximum of these values is used as the impact assessment for the security issue under the premise that attackers may pursue the weakness path that will result in the maximum possible damage to the system.

2.6. Prioritizing Security Fixes

Following the stages of (a) security issue identification and (b) extraction of attributes for each security issue (potential for remote exploitation; impact on confidentiality, integrity and availability; time required for fixing the vulnerability), the framework executes a security issue prioritization step. The goal of this step is to consider a security budget allocation (which is expressed in terms of working hours), and arrange for assigning portions of the budget to the correction of security issues, in order to minimize the residual risk, i.e., the risk owing to the impact of security issues that cannot be fixed, due to security budget constraints.

The security issue prioritization step is performed using linear programming [40]. More specifically, the framework formulates an integer programming optimization problem, where the optimization goal is the minimization of the residual security risk, whereas the available security budget is modeled as a constraint. The formulation of the integer

programming optimization problem is presented in detail in the following paragraphs. The notations used in the integer programming problem formulation are listed in Table 1.

Table 1. Notations used in the integer programming problem formulation.

Notation	Description
S_i	The ith security issue
N	The number of security issues detected
SB	The available security budget, expressed in available working hours
RE_i	1 if S_i is remotely exploitable, otherwise 0
$I_{conf}(S_i)$	The impact of S_i on confidentiality
$I_{integ}(S_i)$	The impact of S_i on integrity
$I_{avail}(S_i)$	The impact of S_i on availability
$I(S_i)$	The overall impact of S_i on the IoT platform, considering all security dimensions (confidentiality, integrity, and availability)
FT_i	The time needed to fix S_i, expressed in working hours
x_i	An output variable of the problem; x_i is set to *1* if security budget is allocated to fixing S_i, otherwise x_i is set to *0*.

At this stage, we consider that no detailed information is available for the following:

- *The importance of each security dimension, either globally or per specific software deployment*. For instance, a web server may be used to make a public database available, and the integrity of the database records may be deemed more important than the availability of the service, while confidentiality may be considered of low importance (since the database is public). On the contrary, for a web server managing a health record database, all the security dimensions (confidentiality, integrity, and availability) may be deemed of high importance.
- *The importance of each software deployment*. In the previous example, the impact of any demotion of the value of the public database may be deemed to be lower than the impact of a corresponding demotion in the value of the medical record database.

Under the absence of the information listed above, the algorithm will operate under the assumption that (a) all security dimensions are of equal importance and (b) all software deployments are of equal importance. Extensions that will consider potential sources of additional information that will enable the algorithm to take into account variations in the importance of security dimensions and resources are considered part of our future work.

The formulation of the integer programming problem proceeds as follows:

- Firstly, the optimization target is formulated. Since the goal of the optimization is the minimization of the residual risk, which is mapped to the impact of the software vulnerabilities that remain unfixed, the objective function of the integer programming problem is

$$minimize \sum_{i=1}^{N} RE_i * (1 - x_i) * I(S_i) \quad (2)$$

where $I(S_i)$ is the security impact of vulnerability S_i. The impact can be directly drawn from the vulnerability impact assessment step, as detailed in Section 2.5; if the vulnerability impact assessment step produces separate assessments of the vulnerability on the different security dimensions (confidentiality, integrity, and availability), the overall impact of the vulnerability can be estimated as the sum of its individual impact on each security dimension, i.e.,:

$$I(S_i) = I_{conf}(S_i) + I_{integ}(S_i) + Ii, avail(S_i) \quad (3)$$

Note that in Equation (2) the impact of each vulnerability $impact(S_i)$ is multiplied (a) by the quantity RE_i, positioning it so that only the impact of remotely exploitable

vulnerabilities is considered, and (b) by the quantity $(1 - x_i)$, positioning it so that only the impact of security issues that remain unfixed is taken into account.

- Subsequently, the cost of implementing fixes to the security issues is calculated using the formula

$$Cost = \sum_{i=1}^{N} x_i * FT_i \tag{4}$$

  In Equation (4) the cost FT_i of fixing a security issue S_i is multiplied by variable x_i positioning it so that only the cost of fixes that are selected to be applied is considered.
- Finally, the security budget constraint is applied, which is formulated as follows:

$$Cost \leq SB \tag{5}$$

  The solution of the integer programming optimization problem is a set of value assignments to variables x_i

$$\begin{gathered} Solution = \{x_1 = v_1, x_2 = v_2, \ldots, x_N = v_N\} \\ \text{where } v_i = \begin{cases} 1 & \text{if security issue } S_i \text{ is selected to be fixed,} \\ 0 & otherwise \end{cases} \end{gathered} \tag{6}$$

3. Results

To validate our approach, we conducted experiments to (a) provide a proof of concept for the proposed software vulnerability management framework and (b) gain insight on the quality of the vulnerability prioritization recommendations produced by the framework.

In these experiments, we used the following software components and platform setups:

1. Software components implementing the five most widely used technologies in IoT networks [41], both individually and as elements of an IoT platform;
2. An indicative small office/home office (SOHO) configuration.

Since, to the best of our knowledge, no commercial system or research proposal offers prioritization of addressing the vulnerability software components, considering the impact of each vulnerability, the associated technical debt for its remediation, and the available security budget, the following baselines were used to assess the effectiveness of the prioritization mechanism:

1. *BCMM*, i.e., an approach according to which vulnerabilities are processed according to the characterization assigned by the SonarQube analysis [42], and more specifically **b**locker vulnerabilities are handled first, followed by vulnerabilities characterized as **c**ritical, **m**ajor, and **m**inor, in that order. Considering that for the vulnerabilities reported by Sonar, only the technical debt (fix time) is available, three variants of BCMM are considered, namely (a) $BCMM_{SFT}$, where within each categorization, vulnerabilities with the shortest fix time are handled first, (b) $BCMM_{LFT}$, where within each categorization vulnerabilities with the largest technical debt are handled first, and (c) $BCMM_{Rand}$, where vulnerabilities within each categorization are considered in random order.
2. *IMM*: According to the descriptions given for SonarQube vulnerability characterizations [42], *blocker* issues are bugs with a high probability to impact the behavior of the application in production, and should be fixed immediately and *critical* issues are bugs with a low probability to impact the behavior of the application in production or issues that represent security flaw vulnerabilities and must also be fixed immediately. Since both classes are designated to require immediate handling, in the *IMM* approach they are merged to a single class, **i**mmediate, while issues in the **m**ajor and **m**inor classes are retained in their original characterization. Similarly to the *BCMM* approach, three variants are considered, namely IMM_{SFT}, IMM_{LFT}, and IMM_{Rand}.

3.1. Experiments for the Commonly Used IoT Technologies

In this subsection we present our experiments concerning a configuration which comprises the five most widely used technologies in the IoT. As reported by [41], the five most widely used technologies in IoT networks are:

1. The *advanced message-queuing protocol* (AMQP), an open standard protocol used for message exchange, including publish/subscribe and point-to-point, as well as queues [43],
2. *Bluetooth and Bluetooth low-energy* (BLE), a short-range communication protocol and its low-energy variant [44],
3. *Cellular communications*, i.e., implementation of communication through cellular telephony netowrks (2G, 3G, 4G/LTE, and 5G),
4. The *constrained application protocol* (CoAP) [45], a specialized internet protocol for devices with constrained resources (e.g., wireless sensors), which enables both (a) pairs or groups of devices running CoAP and (b) devices running CoAP and the internet.
5. The *data distribution service* for real-time systems (DDS) [46], a networking middleware for realtime systems specified by the object management group (OMG), realizing data-centric publish-subscribe mechanisms which can be easily integrated in the application layer.

Following these data, an IoT system configuration was formulated, running instances of software implementing the above listed technologies as follows:

- AMQP was implemented using RabbitMQ v. 3.4.0 (https://github.com/rabbitmq/rabbitmq-server/tree/rabbitmq_v3_4_0, accessed on 14 May 2023),
- Bluetooth/Bluetooth LE was implemented using the Android 13 drivers (https://android.googlesource.com/kernel/msm/+/refs/tags/android-13.0.0_r0.1/drivers/bluetooth/, accessed on 14 May 2023)
- Cellular communications were implemented using Open5GS v. 2.1.3 (https://github.com/open5gs/open5gs/releases/tag/v2.1.3, accessed on 14 May 2023)
- CoAP was implemented using the CoAP library in Arm Mbed OS 5.14.0 (https://github.com/ARMmbed/mbed-os/releases/tag/mbed-os-5.14.0, accessed on 14 May 2023)
- DDS was implemented using OpenDDS v. 3.16.1 (https://github.com/OpenDDS/OpenDDS/releases/tag/DDS-3.16.1, accessed on 14 May 2023)

The software listed above was analyzed and was found to entail 47 vulnerabilities, accounting for a total technical debt of 967 min, with an overall risk equal to 273.34. In the prioritization experiments we considered the security budget values of 50, 125, 250, and 500 min. Table 2 lists indicative results from applying the security issue prioritization method described in Section 2.6 to the identified security issues. As shown in Table 3, the recommendation in all cases consumes (almost) all the available budget, and manages to effectively direct the available budget to the mitigation of the issues having the highest impact, since the percentage of the total impact mitigated in the recommendation list is consistently higher than the ratio of the available budget to the total technical debt.

Table 2. Results of applying the security issue prioritization to the commonly used IoT technologies' configuration.

Security Budget	# Issues Mitigated	Consumed Budget	Impact Mitigated	% of Total Budget Available	% Issues Mitigated	% Impact Mitigated
50	8	47	58.95	5.17%	17.02%	21.57%
125	15	122	98.25	12.93%	12.62%	35.94%
250	23	247	144.13	25.85%	48.94%	52.73%
500	33	497	210.52	51.71%	70.21%	77.02%

Table 3. Results of applying the security issue prioritization to the SOHO configuration.

Security Budget	# Issues Mitigated	Consumed Budget	Impact Mitigated	% of Total Budget Available	% Issues Mitigated	% Impact Mitigated
250	13	245	107.46	8.63%	13%	14.88%
500	22	500	185.899	17.61%	22%	25.74%
1000	38	980	308.05	34.51%	38%	42.65%
1500	55	1490	437.78	52.46%	55%	60.61%

The results in Table 2 demonstrate that especially when the security budget is very limited, the scarce resources can be efficiently allocated to the mitigation of security issues with a very high impact, which is testified by the fact that for the case of having a security budget equal to 50, the ratio of the mitigated impact percentage is 21.57%, approximately 4 times higher than the percentage of the available budget. When the available security budget increases, this performance margin narrows, declining to the value of approximately 1.5 times higher when the security budget is equal to 500 (or 51.71% of the total available budget).

Figure 2 depicts the effectiveness of the proposed approach against the baseline algorithms concerning the configuration including the commonly used IoT technologies. We can observe that under all security budgets, the proposed approach achieves the highest percentage of the mitigated impact, with a margin ranging from 0.51% to 23.14% against the runner-up (which is the IMM_{SFT} algorithm in all cases) when comparing absolute magnitudes (i.e., $impactMitigated(proposed) - impactMitigated(baseline)$); when considering relative improvements (i.e., $\frac{impactMitigated(proposed) - impactMitigated(baseline)}{impactMitigated(baseline)}$), the effectiveness margin of the proposed algorithm ranges from 0.87% to 12.35%. When the security budget is very small (50, i.e., approximately 5% of the total technical debt), the performance edge of the proposed algorithm is small (relative improvement equal to 0.87%), due to the fact that many vulnerabilities with small fix times with "blocker" and "critical" characterizations have high impacts; hence the IMM_{SFT} algorithm is circumstantially led to "close to optimal" decisions, due to the combined distribution of the impact, criticality level, and technical debt of the software vulnerability dataset.

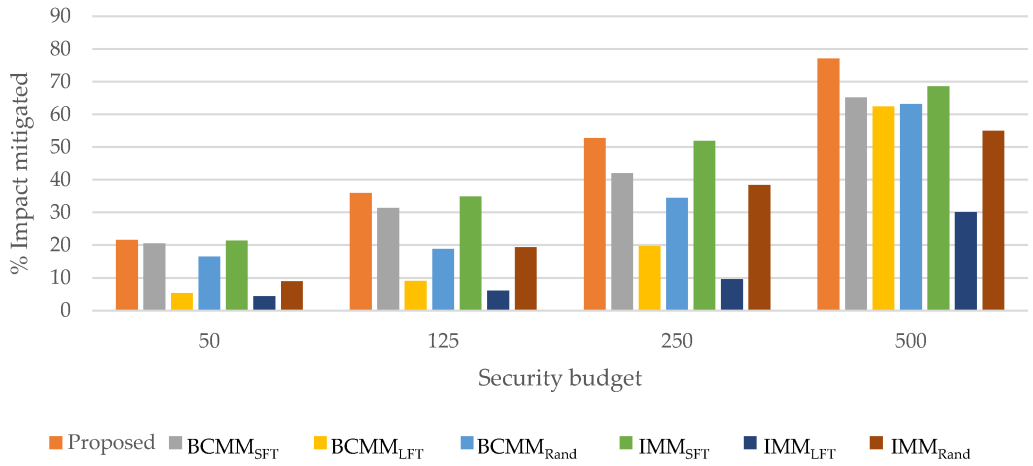

Figure 2. Effectiveness of security issue fix prioritization algorithms for the commonly used IoT technologies' configuration.

This distribution is depicted in Figure 3; in this figure, we can partition vulnerabilities into four quartiles, with Q1 including vulnerabilities with a low cost to fix and their remediation results to high gains in the residual risk of the overall configuration. The presence of numerous "blocker" and "critical" vulnerabilities in this quartile is the reason that leads to the "close to optimal" performance of the IMM_{SFT} for the constrained security budget.

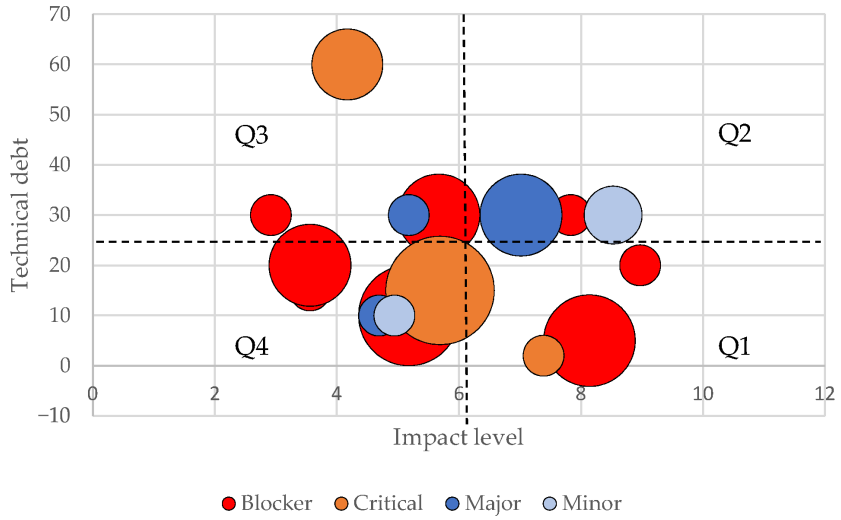

Figure 3. Distribution of the impact, criticality level, and technical debt of the software vulnerabilities in the "commonly used IoT technologies" configuration.

The margin between the proposed approach and other algorithms is larger, ranging from 2.72% to 32.40% in absolute magnitudes, while the corresponding relative improvement ranges from 4.84% to 496.18%. This performance margin is attributed to the capability of the proposed algorithm to direct the security budget to issues whose fixing will lead to the largest reductions in the residual risk. The "largest fix times first" approach again produces the worst results, because each fix applied consumes a large amount of security budget, leading to its depletion without necessarily achieving a respectively high reduction of the residual risk.

3.2. Experiments for the SOHO Configuration

In this subsection we present our experiments concerning a small office/home office (SOHO) configuration, whose architecture follows the typical SOHO architectural style, i.e., the platform a is configured as a "flat" network, where network connectivity is realized by a single device acting both as (a) a layer 2 switch accommodating both wired and wireless protocols for internal nodes (which are few), and (b) as a router providing internet connectivity [47,48]. The specific topology used in the experiment comprises:

1. A router running PFsense (https://github.com/pfsense/pfsense/tree/a81a848e7565cf4b5e1679fe6d08c39d13ab7a6f, accessed on 14 May 2023),
2. A NAS appliance running the Minnow Server (https://github.com/RealTimeLogic/MinnowServer, accessed on 14 May 2023)
3. A smart air-conditioning appliance running the Pymodbus software (https://github.com/pymodbus-dev/pymodbus, accessed on 14 May 2023)
4. A mobile phone and a PC which include Modbus4j software (https://github.com/MangoAutomation/modbus4j, accessed on 14 May 2023) in order to control the air-conditioning appliance.

The SOHO topology corresponding to this configuration is illustrated in Figure 4.

The source code for the software components listed above was collected and processed through the software vulnerability management framework pipeline illustrated in Figure 1 and detailed in Sections 2.1–2.6. In the security fix prioritization step, multiple values were used for the security budget to gain insight on the effect of this parameter in the recommendations formulated, regarding the list of software issues to be mitigated.

The analysis of the software identified 100 security issues with the code, with a total impact equal to 722.24 and an estimated technical debt equal to 2840 min, i.e., a security budget of 2840 developer working minutes is required to mitigate all security issues. Table 3 lists indicative results from applying the security issue prioritization method described

in Section 2.6 to the identified security issues. As shown in Table 3, the recommendation in all cases consumes (almost) all the available budget, and manages to effectively direct the available budget to the mitigation of the issues having the highest impact, since the percentage of the total impact mitigated in the recommendation list is consistently higher than the ratio of the available budget to the total technical debt.

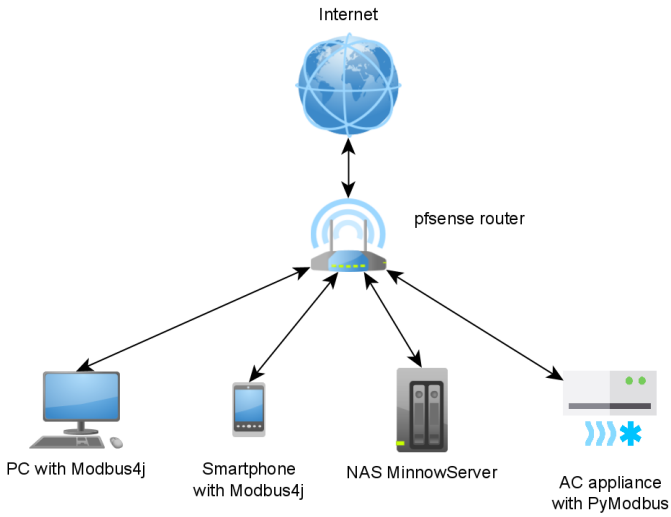

Figure 4. SOHO topology used in the evaluation (adapted from [48]).

In Table 3, we can also notice that especially when the security budget is limited, the constrained resources can be efficiently allocated to the mitigation of security issues with a very high impact, which is demonstrated by the fact that for the case of having a security budget equal to 250, the ratio of the mitigated impact percentage is 1.72 times higher than the percentage of the total available budget. When the available security budget increases, this performance margin narrows, declining to the value of 1.15 times when the security budget is equal to 1500 (or 52.46% of the total available budget).

Figure 5 illustrates the effectiveness of the proposed approach against the baseline algorithms. We can observe that under all security budgets, the proposed approach achieves the highest percentage of mitigated impact, with a margin ranging from 1.42% to 6.78% against the runner-up (which is the IMM_{SFT} algorithm in all cases) when comparing absolute magnitudes (i.e., $impactMitigated(proposed) - impactMitigated(baseline)$); when considering relative improvements (i.e., $\frac{impactMitigated(proposed) - impactMitigated(baseline)}{impactMitigated(baseline)}$), the effectiveness margin of the proposed algorithm ranges from 10.63% to 12.59%. The margin between the proposed approach and other algorithms is larger, ranging from 6.15% to 12.73% in absolute magnitudes, while the relative improvement ranges from 14.06% to 88.45%. This performance margin is again attributed to the capability of the proposed algorithm to direct the security budget to issues whose fixing will lead to the largest reductions in the residual risk. The "largest fix times first" approach again produces the worst results, because each fix applied consumes a large amount of security budget, leading to its depletion, without necessarily achieving a, respectively, high reduction of the residual risk.

Detailed information on the security issues identified, the integer programming problem formulations used for the prioritization of security issue mitigations and the solutions of these problems is available at https://github.com/costasvassilakis/vulnerability-management-framework, accessed on 14 May 2023.

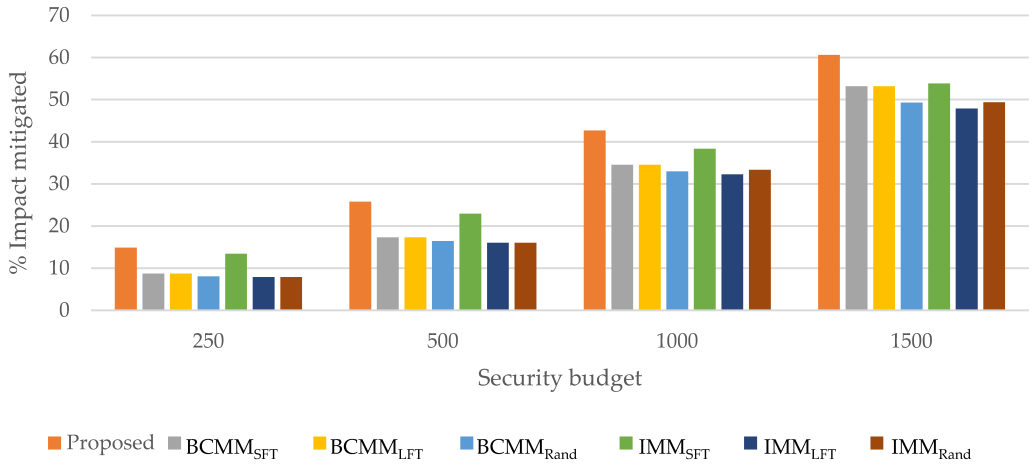

Figure 5. Effectiveness of security issue fix prioritization algorithms for the SOHO configuration.

4. Discussion

The framework presented in this paper and the methods proposed to realize the constituent components of the related workflow can be used by practitioners and researchers alike.

As far as the practical implications of this work are concerned, the proposed framework can be used as an aid in IoT platform implementation by software architects, developers, and security experts, supporting the task of minimizing the overall residual risk. The proposed framework may be also utilized to support different code development and maintenance tasks; for instance, issues of type *BUG* can be extracted from the results of the analysis performed by SAST tools, and the prioritization step could be applied to the bug list in order to guide developers in addressing bugs, or—more generally—improving code quality. Under the design-by-contract approach [49,50], the proposed framework may automatically perform security assessments of alternative implementations of the same contract, and automatically bundle into the executable the implementation providing higher security levels, or issue relevant recommendations to the developers.

In the research domain, starting from the proposed framework, a number of aspects may be further analyzed and elaborated on. THe first topic that can be explored is the exploitation of attack graphs [51,52] to fully consider all possible attack paths to the IoT infrastructure and their repercussions, since remote attacks with low impacts may act as stepping stones for additional attacks that may expose the infrastructure to more serious adverse effects, e.g., by combining a remotely exploitable attack offering to the attacker low privileges with an attack that can be executed locally only and offers privilege escalation [53].

Static security code analysis may be complemented with dynamic security code analysis [54] to uncover a complex flaws or vulnerabilities that cannot be identified by SAST tools due to their perplexed nature.

The estimation of the impact of security issues may be further refined by considering the similarity of the code entailing each particular security issue with code fragments that are rooted to the same weakness and for which the impact has been assessed by human experts. Approaches that assess the functional similarity of code [55] may be used to that effect.

Concerning the prioritization of security issue fixing, a number of extensions are envisioned. The current algorithm assumes the equal importance of all resources in the IoT platform, however this may not hold in all environments; for instance, in the example presented in Section 3, the air-conditioning appliance can be deemed of lower importance than the PC or the NAS appliance, hence software issues with the software running in the air-conditioning appliance may be assigned a lower fixing priority. The business importance of each appliance could be provided externally by human experts and subsequently be considered by the algorithm. Similarly, the prioritization algorithm could be extended to

accommodate diverse importance and could be applied to different security dimension of each appliance or resource: for instance, a public information database may have low requirements for confidentiality and high requirements for integrity and availability, while a health record database would have high requirements for all security dimensions.

Finally, this work may be adapted and used in numerous areas of the software engineering domain, including performance/stress testing and identification of hotspots, testing in different deployment environments, etc.

5. Conclusions

In this paper, we have presented a software vulnerability management framework which supports all the stages of a pipeline for the management of IoT platform software vulnerabilities. More specifically, the framework supports (a) the configuration of software to include only the necessary features, (b) the execution of security-related tests and the compilation of platform-wide software vulnerability lists, (c) the estimation of the impact and the associated fixing cost for each vulnerability, and (d) the prioritization of vulnerability addressing (considering the impact of each vulnerability) the associated technical debt for its remediation and the available security budget.

The work presented in this paper advances the state-of-the-art by (i) proposing a statistics-based method for the estimation of impact of detected vulnerabilities, (ii) proposing an integer programming-based algorithm for prioritizing security fixes with the goal of minimizing the residual risk level, and (iii) proposing a comprehensive framework for the security analysis of platform software which formulates proposals on the prioritization of security issue addressing, taking into account all the aspects (a)–(d) listed in the previous paragraph.

Our future work will focus on the incorporation of dynamic security code analysis and attack graphs into the workflow, as well as the refinement of vulnerability impact estimation.

Author Contributions: Conceptualization, P.S., C.-M.M., C.V. and N.K.; methodology, P.S., C.-M.M., C.V. and N.K.; software, P.S., C.-M.M., C.V. and N.K.; validation, P.S., C.-M.M., C.V. and N.K.; investigation, P.S., C.-M.M., C.V. and N.K.; data curation, P.S., C.-M.M., C.V. and N.K.; writing—original draft preparation, P.S., C.-M.M., C.V. and N.K.; writing—review and editing, P.S., C.-M.M., C.V. and N.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement no. 833673. The work reflects only the authors' view and the Agency is not responsible for any use that may be made of the information it contains.

Informed Consent Statement: Not applicable.

Data Availability Statement: The results from applying the framework processing pipeline on the software of the SOHO configuration listed in Section 3 are available at https://github.com/costasvassilakis/vulnerability-management-framework/tree/main, accessed on 14 May 2023.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Grau, A.; Indri, M.; Bello, L.L.; Sauter, T. Industrial robotics in factory automation: From the early stage to the Internet of Things. In Proceedings of the IECON 2017—43rd Annual Conference of the IEEE Industrial Electronics Society, Beijing, China, 29 October–1 November 2017. [CrossRef]
2. Grau, A.; Indri, M.; Bello, L.L.; Sauter, T. Robots in Industry: The Past, Present, and Future of a Growing Collaboration With Humans. *IEEE Ind. Electron. Mag.* **2021**, *15*, 50–61. [CrossRef]
3. Barai, G.R.; Krishnan, S.; Venkatesh, B. Smart metering and functionalities of smart meters in smart grid—A review. In Proceedings of the 2015 IEEE Electrical Power and Energy Conference (EPEC), London, ON, Canada, 26–28 October 2015. [CrossRef]
4. Coppola, R.; Morisio, M. Connected Car. *ACM Comput. Surv.* **2016**, *49*, 1–36. [CrossRef]
5. Hussain, R.; Zeadally, S. Autonomous Cars: Research Results, Issues, and Future Challenges. *IEEE Commun. Surv. Tutor.* **2019**, *21*, 1275–1313. [CrossRef]

6. Number of Internet of Things (IoT) Connected Devices Worldwide from 2019 to 2021, with Forecasts from 2022 to 2030. 2022. Available online: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/ (accessed on 4 February 2023).
7. The Internet of Things: A Movement, Not a Market. 2017. Available online: https://cdn.ihs.com/www/pdf/IoT-ebook.pdf (accessed on 4 February 2023).
8. binti Mohamad Noor, M.; Hassan, W.H. Current research on Internet of Things (IoT) security: A survey. *Comput. Netw.* **2019**, *148*, 283–294. [CrossRef]
9. Ali, R.F.; Muneer, A.; Dominic, P.D.D.; Taib, S.M.; Ghaleb, E.A.A. Internet of Things (IoT) Security Challenges and Solutions: A Systematic Literature Review. In *Communications in Computer and Information Science*; Springer: Singapore, 2021; pp. 128–154. [CrossRef]
10. HaddadPajouh, H.; Dehghantanha, A.; Parizi, R.M.; Aledhari, M.; Karimipour, H. A survey on internet of things security: Requirements, challenges, and solutions. *Internet Things* **2021**, *14*, 100129. [CrossRef]
11. Omolara, A.E.; Alabdulatif, A.; Abiodun, O.I.; Alawida, M.; Alabdulatif, A.; Alshoura, W.H.; Arshad, H. The internet of things security: A survey encompassing unexplored areas and new insights. *Comput. Secur.* **2022**, *112*, 102494. [CrossRef]
12. Evaluators, I.S. SOHOpelessly Broken 2.0. 2019. Available online: https://www.ise.io/casestudies/sohopelessly-broken-2-0/ (accessed on 4 February 2023).
13. Herwig, S.; Harvey, K.; Hughey, G.; Roberts, R.; Levin, D. Measurement and Analysis of Hajime, a Peer-to-peer IoT Botnet. In Proceedings of the 2019 Network and Distributed System Security Symposium, San Diego, CA, USA, 24–27 February 2019; Internet Society: Reston, VA, USA, 2019. [CrossRef]
14. Bastos, G.; Marzano, A.; Fonseca, O.; Fazzion, E.; Hoepers, C.; Steding-Jessen, K.; Chaves, C.M.H.P.C.; Cunha, I.; Guedes, D.; Meira, W. Identifying and Characterizing Bashlite and Mirai C&C Servers. In Proceedings of the 2019 IEEE Symposium on Computers and Communications (ISCC), Barcelona, Spain, 29 June–3 July 2019. [CrossRef]
15. Hiesgen, R.; Nawrocki, M.; Schmidt, T.C.; Wählisch, M. The Race to the Vulnerable: Measuring the Log4j Shell Incident. *arXiv* **2022**, arXiv:2205.02544.
16. OpenLiberty Group. Open Liberty. 2023. Available online: https://openliberty.io/ (accessed on 4 February 2023).
17. OpenLiberty Group. Open Liberty: Feature Overview. 2023. Available online: https://openliberty.io/docs/latest/reference/feature/feature-overview.html (accessed on 4 February 2023).
18. Sotiropoulos, P.; Vassilakis, C. The additional testsuite framework: Facilitating software testing and test management. *Int. J. Web Eng. Technol.* **2022**, *17*, 296–334. [CrossRef]
19. Al-boghdady, A.; Wassif, K.; El-ramly, M. The presence, trends, and causes of security vulnerabilities in operating systems of iot's low-end devices. *Sensors* **2021**, *21*, 2329. [CrossRef] [PubMed]
20. Kaur, A.; Nayyar, R. A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code. *Procedia Computer Science* **2020**, *171*, 2023–2029. [CrossRef]
21. OWASP. *OWASP Code Review Guide v2*; Technical Report; OWASP: Wakefield, MA, USA, 2017.
22. Mathas, C.M.; Vassilakis, C.; Kolokotronis, N.; Zarakovitis, C.C.; Kourtis, M.A. On the design of IoT security: Analysis of software vulnerabilities for smart grids. *Energies* **2021**, *14*, 2818. [CrossRef]
23. Schiller, E.; Aidoo, A.; Fuhrer, J.; Stahl, J.; Ziörjen, M.; Stiller, B. Landscape of IoT security. *Comput. Sci. Rev.* **2022**, *44*, 100467. [CrossRef]
24. Calatayud, B.M.; Meany, L. A comparative analysis of Buffer Overflow vulnerabilities in High-End IoT devices. In Proceedings of the 2022 IEEE 12th Annual Computing and Communication Workshop and Conference, CCWC 2022, Las Vegas, NV, USA, 26–29 January 2022; pp. 694–701. [CrossRef]
25. de Vicente Mohino, J.; Higuera, J.B.; Higuera, J.R.B.; Montalvo, J.A.S. The Application of a New Secure Software Development Life Cycle (S-SDLC) with Agile Methodologies. *Electronics* **2019**, *8*, 1218. [CrossRef]
26. SAFECode. *Fundamental Practices for Secure Software Development*; Technical Report 3rd; SAFEcode: Wakefield, MA, USA, 2018.
27. Rashid, A.; Chivers, H.; Danezis, G.; Lupu, E.; Martin, A. *CyBok Version 1.0*; Technical Report; CyBok: Bristol, UK, 2019.
28. Dewhurst, R. *OWASP Static Code Analysis*; Technical Report; OWASP: Wakefield, MA, USA, 2023.
29. Sachidananda, V.; Bhairav, S.; Ghosh, N.; Elovici, Y. PIT: A Probe Into Internet of Things by Comprehensive Security Analysis. In Proceedings of the 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), Rotorua, New Zealand, 5–8 August 2019; pp. 522–529. [CrossRef]
30. Samtani, S.; Yu, S.; Zhu, H.; Patton, M.; Chen, H. Identifying SCADA vulnerabilities using passive and active vulnerability assessment techniques. In Proceedings of the 2016 IEEE Conference on Intelligence and Security Informatics (ISI), Tucson, AZ, USA, 28–30 September 2016; pp. 25–30. [CrossRef]
31. Geneiatakis, D.; Kounelis, I.; Neisse, R.; Nai-Fovino, I.; Steri, G.; Baldini, G. Security and privacy issues for an IoT based smart home. In Proceedings of the 2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 22–26 May 2017; pp. 1292–1297. [CrossRef]
32. Overstreet, D.; Wimmer, H.; Haddad, R.J. Penetration Testing of the Amazon Echo Digital Voice Assistant Using a Denial-of-Service Attack. In Proceedings of the 2019 SoutheastCon, Huntsville, AL, USA, 11–14 April 2019; pp. 1–6. [CrossRef]
33. He, D.; Yu, X.; Li, T.; Chan, S.; Guizani, M. Firmware Vulnerabilities Homology Detection Based on Clonal Selection Algorithm for IoT Devices. *IEEE Internet Things J.* **2022**, *9*, 16438–16445. [CrossRef]

34. Kotenko, I.; Izrailov, K.; Buinevich, M. Static Analysis of Information Systems for IoT Cyber Security: A Survey of Machine Learning Approaches. *Sensors* **2022**, *22*, 1335. [CrossRef] [PubMed]
35. Akhilesh, R.; Bills, O.; Chilamkurti, N.; Chowdhury, M.J.M. Automated Penetration Testing Framework for Smart-Home-Based IoT Devices. *Future Internet* **2022**, *14*, 276. [CrossRef]
36. Zheng, Y.; Li, Y.; Zhang, C.; Zhu, H.; Liu, Y.; Sun, L. Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual, 18–22 July 2022; pp. 417–428. [CrossRef]
37. Prehofer, C. Feature-oriented programming: A new way of object composition. *Concurr. Comput. Pract. Exp.* **2001**, *13*, 465–501. [CrossRef]
38. Zave, P. Requirements for evolving systems: A telecommunications perspective. In Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, Toronto, ON, Canada, 27–31 August 2001. [CrossRef]
39. Apel, S.; Batory, D.; Kästner, C.; Saake, G. *Feature-Oriented Software Product Lines*; Springer: Berlin/Heidelberg, Germany, 2013. [CrossRef]
40. Dantzig, G.B. Linear Programming. *Oper. Res.* **2002**, *50*, 42–47. [CrossRef]
41. TechTarget. Top 12 Most Commonly Used IoT Protocols and Standards. 2022. Available online: https://www.techtarget.com/iotagenda/tip/Top-12-most-commonly-used-IoT-protocols-and-standards (accessed on 14 May 2023).
42. SonarQube. Issues. 2023. Available online: https://docs.sonarqube.org/latest/user-guide/issues/ (accessed on 14 May 2023).
43. AMQP group AMQP v1.0. 2011. Available online: https://www.amqp.org/sites/amqp.org/files/amqp.pdf (accessed on 14 May 2023).
44. Heydon, R. *Bluetooth Low Energy*; Prentice Hall: Philadelphia, PA, USA, 2012.
45. Bormann, C.; Castellani, A.P.; Shelby, Z. CoAP: An Application Protocol for Billions of Tiny Internet Nodes. *IEEE Internet Comput.* **2012**, *16*, 62–67. [CrossRef]
46. Yang, J.; Sandstrom, K.; Nolte, T.; Behnam, M. Data Distribution Service for industrial automation. In Proceedings of the 2012 IEEE 17th International Conference on Emerging Technologies &amp Factory Automation (ETFA 2012), Krakow, Poland, 17–21 September 2012. [CrossRef]
47. IPCisco. Small Office/Home Office (SOHO) Architecture. 2018. Available online: https://ipcisco.com/lesson/network-topology-architectures/ (accessed on 14 May 2023).
48. Penz, R. Ready Your Home Network for IoT. 2016. Available online: https://robert.penz.name/1341/ready-your-home-network-for-iot/ (accessed on 14 May 2023).
49. Ozkaya, M. Teaching Design-by-Contract for the Modeling and Implementation of Software Systems. In Proceedings of the 14th International Conference on Software Technologies, Prague, Czech Republic, 26–28 July 2019; SCITEPRESS—Science and Technology Publications: Setúbal, Portugal, 2019. [CrossRef]
50. Silva, C.; Guérin, S.; Mazo, R.; Champeau, J. Contract-based design patterns. In Proceedings of the 15th International Conference on Availability, Reliability and Security, Virtual, 25–28 August 2020. [CrossRef]
51. Wang, B.; Gong, N.Z. Attacking Graph-based Classification via Manipulating the Graph Structure. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019. [CrossRef]
52. Ghazo, A.T.A.; Ibrahim, M.; Ren, H.; Kumar, R. A2G2V: Automatic Attack Graph Generation and Visualization and Its Applications to Computer and SCADA Networks. *IEEE Trans. Syst. Man Cybern. Syst.* **2020**, *50*, 3488–3498. [CrossRef]
53. O'Leary, M. Privilege Escalation in Linux. In *Cyber Operations*; Apress: Berlin, Germany, 2019; pp. 419–453. [CrossRef]
54. Rangnau, T.; Buijtenen, R.v.; Fransen, F.; Turkmen, F. Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines. In Proceedings of the 2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC), Eindhoven, The Netherlands, 5–8 October 2020. [CrossRef]
55. Zhao, G.; Huang, J. DeepSim: Deep learning code functional similarity. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, FL, USA, 4–9 November 2018. [CrossRef]