computers

MDPI

Article

Design and Implementation of SFCI: A Tool for Security Focused Continuous Integration

Michael Lescisin [1,*], Qusay H. Mahmoud [1,*] and Anca Cioraca [2]

[1] Department of Electrical, Computer and Software Engineering; Ontario Tech University; Oshawa, ON L1G 0C5, Canada
[2] GE Grid Solutions, Markham, ON L6C 0M1, Canada; anca.cioraca@ge.com
* Correspondence: michael.lescisin@ontariotechu.net (M.L.); qusay.mahmoud@uoit.ca (Q.H.M.)

Received: 23 September 2019; Accepted: 30 October 2019; Published: 1 November 2019

Abstract: Software security is a component of software development that should be integrated throughout its entire development lifecycle, and not simply as an afterthought. If security vulnerabilities are caught early in development, they can be fixed before the software is released in production environments. Furthermore, finding a software vulnerability early in development will warn the programmer and lessen the likelihood of this type of programming error being repeated in other parts of the software project. Using Continuous Integration (CI) for checking for security vulnerabilities every time new code is committed to a repository can alert developers of security flaws almost immediately after they are introduced. Finally, continuous integration tests for security give software developers the option of making the test results public so that users or potential users are given assurance that the software is well tested for security flaws. While there already exists general-purpose continuous integration tools such as Jenkins-CI and GitLab-CI, our tool is primarily focused on integrating third party security testing programs and generating reports on classes of vulnerabilities found in a software project. Our tool performs all tests in a snapshot (stateless) virtual machine to be able to have reproducible tests in an environment similar to the deployment environment. This paper introduces the design and implementation of a tool for security-focused continuous integration. The test cases used demonstrate the ability of the tool to effectively uncover security vulnerabilities even in open source software products such as ImageMagick and a smart grid application, Emoncms.

Keywords: analysis tools; continuous integration; software security; testing

1. Introduction

As an increasing amount of critical systems becomes more and more software orientated, the problem of software security vulnerabilities becomes more and more of a concern. The Internet of Things (IoT) and the smart grid are two manifestations of software systems playing a large role in daily life. Software written for these systems needs to be as secure as possible to ensure safe and reliable operation. There have been many proposed solutions that strive to make the likelihood of exploitable software vulnerabilities as low as possible. Qubes OS founder Joanna Rutkowska categorizes these approaches into three broad categories, security by correctness, security by isolation, and security by obscurity (randomization) [1]. Security by correctness refers to ensuring that all program source code is bug free. Some of the solutions for security by correctness include static code analysis (reading through program source code and searching for potential bugs), dynamic program analysis (executing the program with a wide variety of inputs to verify that it behaves securely), and safe programming languages. Security by isolation refers to confining an application so that it is only allowed to interact with resources with which it has been given permission to interact. The principal

of least privilege is an example of security by isolation. Security by obscurity (randomization) refers to making existing security vulnerabilities difficult to exploit. Address Space Layout Randomization (ASLR) is an example of security by obscurity as it attempts to make memory locations of critical data difficult to find if a memory access violation occurred from a security vulnerability such as a buffer overflow. This paper is only concerned with the approach of security by code correctness, as our tool is only useful for presenting information about security vulnerabilities in a developer's code, as opposed to making them more difficult to exploit or limiting the damage they could cause if they were exploited. This is not to say that security by code correctness, or at least security by code improvement, is superior to the other approaches, but rather, that in applications such as the Internet of Things (IoT) and the smart grid, ensuring that programs behave as they are defined [2] (i.e., a smart grid electrical power distribution network should have almost 100% uptime) is equally critical, if not more, than ensuring that users are only confined to their allocated resources. The approaches are not mutually exclusive, as even a system with strong isolation, such as Qubes OS, has a small Trusted Computing Base (TCB). This TCB cannot be made more secure through isolation efforts, but could however be checked with static and dynamic program analysis tools to ensure it behaves as defined. The tool described in this paper would be useful for improving the security of any program, regardless if it is or is not considered as part of the TCB. There exists a wide variety of dynamic software analysis tools and penetration testing tools designed for detecting, and sometimes exploiting, security vulnerabilities within a program. These tools cover a wide variety of vulnerabilities including memory safety errors (buffer overflow, use-after-free, etc.), as well as input sanitization issues (SQL injection, command injection, path traversal, etc.). These tools are often used manually by software developers and penetration testers to find security flaws within a program before they are exploited by malicious hackers. Continuous integration is a software development process where developers integrate code into a shared repository several times per day, allowing software development teams to detect problems early [3]. Continuous integration recommends that build automation and automated testing be part of the development pipeline. It also recommends the use of a revision control system. This paper will discuss the design and implementation of a testing and reporting tool for evaluating the security of a program in the context of a continuous integration process. The main contributions of this paper are as follows:

- A tool for running automated security tests against the current state of a project's Git repository and displaying the results in a web format.
- A security test case for ImageMagick that attempts to exploit the CVE-2016-3715 vulnerability and delete an arbitrary file.
- A security test case for ImageMagick that attempts a path traversal exploit in linking bitmap images to SVG images.
- Discovery of a vulnerability in ImageMagick where an SVG file disguised as a raster image such as PNG or JPEG can cause a potential information leak.
- Discovery of a cross-site-request-forgery vulnerability in Open Energy Monitor's Emoncms and a test case for its detection.

The rest of this paper is organized as follows. Sections 2 and 3 discuss the motivation and related work, respectively. The architecture of the tool is presented in Section 4, and the various software packages used are discussed in Section 5. Recommended use of Security Focused Continuous Integration (SFCI) is discussed in Section 6, and evaluation results of using the tools with our projects, as well as third party open source software products are presented and discussed in Sections 7–9. The limitations of SFCI are presented in Section 10, while conclusions and ideas for future work are presented in Section 11.

2. Motivation

Software security is becoming an increasingly important aspect of software development as more and more software is being used to perform critical tasks. The traditional software development model

where software security is considered as a separate entity from regular code development has failed many times and has resulted in the release of vulnerable software. Security vulnerabilities are often introduced into programs when the development team lacks the time or financial resources for proper testing or is simply unaware of potential security pitfalls [4]. An example of this was the 2014 Sony Pictures Entertainment hack. Research has shown that if Sony had implemented even some of the well-known critical controls, the attack would have been stopped or at least greatly hindered [5]. Secure software development can be moved closer to the developers by automating the testing process and providing the developers with frequent, easy-to-understand information about security flaws, which they have introduced into the program. Since its creation in 2005, Git has risen in popularity to become one of the most popular version control systems in use today. This rise in popularity is arguably attributable to web services such as GitHub, which provide users with a well-designed, easy-to-understand interface for browsing a project. Given the proliferation of Git, it is reasonable to assume that most software developers are familiar with it or could learn it quickly thanks to the abundance of documentation and tutorials available online. Similar to how GitHub presents project information in a clean, easy-to-understand manner, a tool of similar simplicity, which integrates with Git, could be very effective at displaying software security information to a software developer. It is this integration with Git that allows our tool to do continuous integration as the Git version control system provides a timeline of software project updates for which each milestone, or commit, can be tested for security issues. With this type of continuous integration tool, even a software developer without a background in software security could see if his/her code is insecure, and therefore learn how to write secure code as he/she continues developing the software project.

3. Related Work

Secure Development Operations (Secure DevOps) have existed with the goal of changing the software development attitude with respect to security by applying DevOps best practices such as: adding automated security testing, standardizing the integration cycle, and introducing security concerns at the inception of projects (proactive), rather than after the fact (reactive) [6]. This section discusses related work and tools implementing Secure DevOps practices and explains how our tool is different.

3.1. BDD-Security

BDD-Security is a security testing framework that uses natural language to describe security requirements as features. These requirements are executable as standard unit/integration tests [7]. BDD-Security uses Cucumber to generate HTML reports on the security status of the application. BDD-Security can be integrated with Jenkins-CI for reporting security regressions [8]. BDD-Security uses Selenium/WebDriver, OWASP ZAP, SSLyze, and Tennable's Nessus scanner for detecting vulnerabilities [9]. BDD-Security tests web applications and APIs from an external point of view and therefore does not require application source code [9]. While BDD-Security provides many features for testing for higher level security vulnerabilities (SQL injections, authentication vulnerabilities, etc.), it does not provide tools for detecting lower level security vulnerabilities such as buffer overflows or use-after-free vulnerabilities.

3.2. Valgrind Plugin: Jenkins

There exists a Valgrind plugin for Jenkins-CI that allows Valgrind's Memcheck to be used in Jenkins-CI tests [10]. This could be used for finding memory safety errors such as buffer overflows or use-after-free errors.

3.3. Zapper Plugin: Jenkins

The Zapper plugin allows a Jenkins-CI user to use OWASP's ZAP for automated security testing of a web application [11].

3.4. Tinfoil Security

Tinfoil Security provides a cloud-based service for testing a web application for security vulnerabilities. It provides security reports for each type of vulnerability for which the scanner searches [12]. There also exists a plugin for Jenkins-CI that integrates Tinfoil Security tests with the Jenkins-CI automated testing system [13].

3.5. Arachni

Arachni is a web application security scanner that performs a wide variety of security checks [14]. Although there does not exist a Jenkins-CI plugin for Arachni, it has been manually integrated with some success [15].

3.6. GitLab-CI

GitLab provides a continuous integration tool for running continuous building and testing of GitLab projects. The architecture of GitLab-CI consists of one GitLab instance and one or more GitLab Runners [16]. The GitLab Runners are the execution environments where a project's code is tested. GitLab Runner supports a variety of execution environments including Docker containers [17]. The GitLab Runners reside on user provided servers and are signalled by GitLab-CI, which instructs the Runners to execute the continuous integration tests and return the results back to GitLab-CI. GitLab-CI then uses this information to generate a report on the testing status of the project. Unlike SFCI, GitLab-CI is not focused on security and only provides the user with the build status (pass/fail), as well as the command line output from running the tests [18]. It displays no detailed information about classes of bugs found in the application.

Our tool, although inspired by the above-mentioned tools, differs from them in several ways. It presents information in a similar manner to the Tinfoil Security cloud service, but unlike the Tinfoil Security cloud service, it is designed to run tests in a virtual machine configured with all services needed for the application in question, as opposed to Tinfoil Security, which only tests live websites. Similar to Tinfoil Security, BDD-Security is designed for security testing of websites, while our tool is for general security testing. Arachni is similar in that it too is only designed for detecting web application vulnerabilities and cannot be applied for detecting lower level (such as memory safety) vulnerabilities. SFCI is not in direct competition with popular CI tools such as Jenkins-CI, but rather, could be used in conjunction with a popular CI tool. SFCI differs from tools such as Jenkins in that there is an emphasis on running all tests in a snapshot VM, primarily for the purpose of reproducibility; we want the test environment to be as similar to the deployment environment as possible. Jenkins-CI can be used for executing tests within a VM [19], but this however is not its usual intended use. SFCI could be integrated with Jenkins-CI by having the build pipeline run SFCI against the repository and then possibly use the results from SFCI for displaying in the Jenkins interface in addition to the SFCI created web report. GitLab-CI provides an architecture of instance and runner similar to our design of host and VM. At this point in time, SFCI is only designed for executing continuous integration tests on a single VM instance running on the same hardware as the report generation processes. In the future, if SFCI is expanded to execute its continuous integration tests on multiple remote VMs, GitLab-CI will be considered as a tool for dispatching and executing the tests.

4. Architecture

Our tool, which is part of a security evaluation framework, is implemented as a directory containing all the sub-programs required for testing and report generation. Therefore, to set up this tool, a user needs to install all dependencies (Python, Jinja2, QEMU, etc.) and then add their project and test cases to our tool's directory. A user is required to create one copy of our tool's directory for every project they wish to test, as shown in Figure 1.

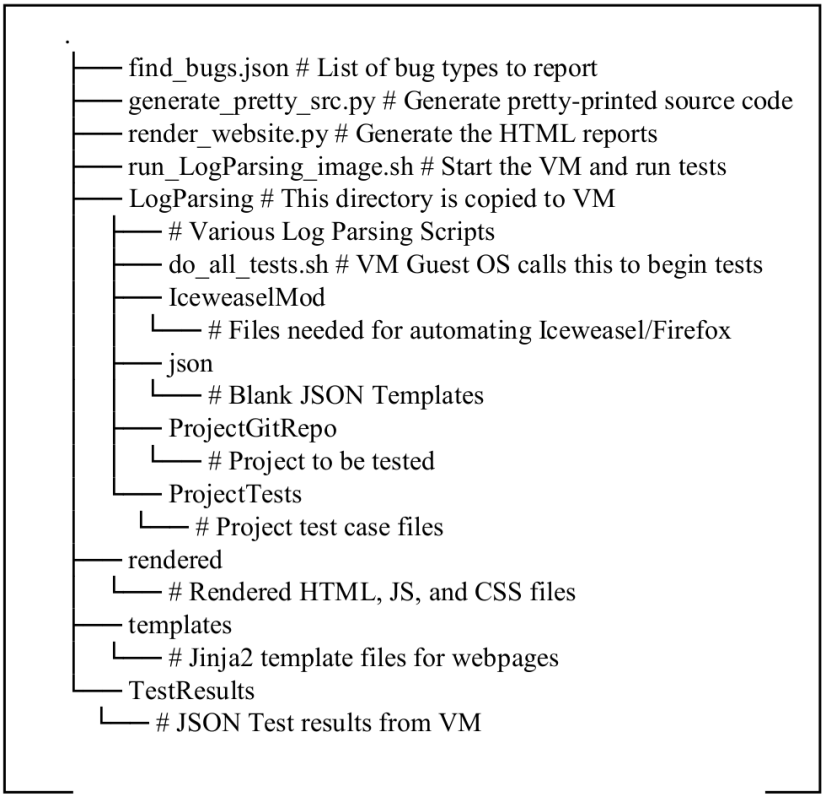

Figure 1. Simplified layout of the files used by our tool.

4.1. Setup Filesystem

The root directory of our tool has a directory named LogParsing. Everything downstream of this directory will be sent to the virtual machine described in the next step. The LogParsing directory has a directory named ProjectGitRepo, and it contains the Git repository of our project that we wish to test. The LogParsing directory also contains a directory named ProjectTests, which contains the test case programs for testing our project. Finally, the LogParsing directory contains the file do_all_tests.sh. This is the file that the virtual machine automatically executes, and it therefore must call all the tests defined in LogParsing/ProjectTests, parse the results with the parsers located in LogParsing/, and send the formatted results back to the host machine.

4.2. Virtual Machine

The virtual machine loads an image of Debian Jessie with all the packages needed for running the tests. In addition, the virtual machine image is configured, by crontab, to execute the do_all_tests.sh script as soon as it starts. The virtual machine is linked to the host via a Linux TAP interface. This allows any TCP/IP based service to work to enable communication between the host and guest as is needed for sending the test results back to the host.

4.3. Parsers and Automation

When the penetration testing tools used by our framework are executed, they are instructed to output their results into a log file. Our framework contains parsers, which parse through these log files and populate JSON files containing information about the vulnerabilities found in the program.

4.4. Web Report Rendering

The JSON test result files are downloaded from the virtual machine through the TAP network interface. These files contain information such as what URL parameters are vulnerable to SQL injection

and which attack strings can exploit them or what input values cause a buffer overflow and where does it occur, for example. The render_website.py script is then executed, which reads all the JSON encoded result files and uses Jinja2 to render the static pages. The generate_pretty_src.py script is then executed, which creates web pages displaying in pretty-printed format the contents of the source code files (located in LogParsing/ProjectGitRepo) and test case files (located in LogParsing/ProjectTests). This is so that a vulnerability description may link to source code files that possess the given vulnerability or test case files that have triggered the vulnerability.

The architecture of our framework (Figure 2) is designed to facilitate the addition of new third-party security testing tools. As discussed in the above sub-sections, as long as a developer is able to programmatically execute the security testing tool and parse its log files to extract meaningful information about the discovered software vulnerabilities, the developer is able to integrate the tool with SFCI. The difficulty of integrating the third party tool is largely dependent on how easily the tool can be automated, as well as the formatting of the output logs that it generates [20].

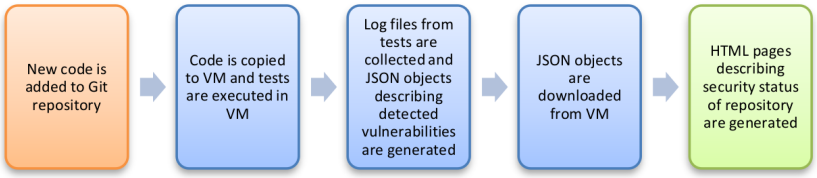

Figure 2. Workflow from code commit to security report generation.

5. Existing Software Packages

Our tool employs several pre-existing software packages for vulnerability detection, version control, and report generation. The tools AddressSanitizer, Valgrind, Sqlmap, Commix, XSS Me, and DotDotPwn were used for detecting vulnerabilities in the software being tested. These tools were selected as they cover the nine types of vulnerabilities reported on by our tool. MITRE listed the top four software vulnerabilities as: SQL injection, OS command injection, buffer overflow, and cross-site scripting. Path traversal vulnerabilities were ranked at number 13. These vulnerabilities can be divided into two broad categories: memory safety vulnerabilities and code injection vulnerabilities. Memory safety vulnerabilities are concerned with illegal accesses to memory (such as buffer overflows or use-after-free), while code injection vulnerabilities are concerned with tricking a victim program to treat untrusted data as code (SQL injection, cross-site-scripting, etc.) [21]. For the memory safety vulnerabilities, two popular testing programs are Valgrind Memcheck and AddressSanitizer. AddressSanitizer is routinely used by Google for testing for memory safety bugs in their Chrome browser and has found over 300 previously-undetected vulnerabilities [22]. Although slower than AddressSanitizer, Valgrind has also been used for bug detection in a wide variety of popular software [23]. Commix, SQLmap, and DotDotPwn were chosen for SQL injection, OS command injection, and path traversal, respectively, as they are found as included packages in the popular penetration testing Linux distribution, Kali Linux [24], and are thus well established tools in the computer security community. Wanting to make our tool capable of automated testing of web pages, we modified the XUL code of Iceweasel (Debian's Firefox) to accept remote privileged JavaScript commands from a testing process. This also gave us the possibility of automating a Firefox plugin of which there are many with the purpose of penetration testing [25]. From these plugins, we chose XSS Me as the default XSS penetration testing tool for our tool. Finally, all our chosen penetration testing tools were free and open-source, which reduced the cost of building our tool and gave us, and the end-user, more freedom to modify any of the programs as needed. Version control was done with Git, and report generation was done with Jinja2. A sandboxed, snapshot (stateless) testing environment was provided by QEMU-KVM. The following discusses these software packages in detail.

5.1. Git

Git was created in 2005 by Linus Torvalds with the purpose of maintaining the development of the Linux kernel [26]. Since then, it has been used to maintain thousands of projects. Various online Git repositories exist, including GitHub, GitLab, and BitBucket. Git has the advantage that everything is stored locally, in the .git directory. This makes adding functionality simple as one only needs to access files in the .git directory.

5.2. Jinja2

Jinja2 is a powerful template engine for Python [27]. In our tool, it is used in the final HTML report generation. Our tool uses Jinja2 to generate static web pages from our template and result files. The static content has the advantage that it can be deployed on any web server and only needs to be rendered one time per commit, as opposed to every time the page is loaded.

5.3. QEMU-KVM

QEMU is a virtual machine hypervisor. It can take advantage of the Kernel Virtual Machine (KVM) provided by the Linux kernel if it is available [28]. It has the feature of running in snapshot mode where no changes will be written back to the virtual hard drive. This gave us the advantage of always starting at the same filesystem state whenever conducting any of our automated tests [29].

5.4. AddressSanitizer

AddressSanitizer is part of the Clang toolchain and is described as a "fast memory error detector" [30]. It adds its instrumentation code to the program at compile-time. It is capable of detecting the following types of memory related bugs:

- Out-of-bounds accesses to heap, stack, and globals
- Use-after-free
- Use-after-return (to some extent)
- Double-free
- Invalid-free
- Memory leaks (experimental)

5.5. Valgrind

Valgrind is a framework for building dynamic analysis tools [31]. Unlike AddressSanitizer, the instrumentation code is added at run-time, and therefore, the programs being testing do not need to be compiled any differently. Two tools that Valgrind comes packaged with are Memcheck and Helgrind. Memcheck is used for finding memory errors, and Helgrind is used for finding data race conditions.

5.6. Sqlmap

Sqlmap is a penetration testing tool for detecting and exploiting SQL injection security flaws [32]. It has been used to successfully detect SQL injection flaws in production software [33,34].

5.7. Commix

Commix (COMMand Injection eXploiter) is an automated tool written by Anastasios Stasinopoulos for testing web based applications for errors related to command injection attacks. It has successfully found several zero-day vulnerabilities in production software [35].

5.8. XSS Me

XSS Me is a tool created by Security Compass for testing websites for cross-site-scripting (HTML injection) vulnerabilities. It comes in the form of a Firefox add-on and can be installed through their GitHub repository [36].

5.9. DotDotPwn

DotDotPwn is a Perl script used for testing a website for path traversal vulnerabilities. Alternatively, it can be instructed to output its path traversal strings to the Standard Output, where it can be used to fuzz test any application for path traversal vulnerabilities [37]. It has successfully found path traversal vulnerabilities in FTP servers [38,39].

Although our tool comes preconfigured with the above-mentioned security testing packages, a user is not confined to only using these programs. There are currently other security testing programs that might be of interest to a software developer for use with SFCI, as well as other security testing programs that may arise in the future. Our Getting Started Guide provides instructions on how to integrate a third-party tool, American Fuzzy Lop, with SFCI [40].

6. Recommended Use

The purpose of SFCI is to protect against the introduction of security vulnerabilities into a software project. This is done in two ways: the first way is checking newly introduced code for popular classes of vulnerabilities (such as command injections), and the second way is preventing the reintroduction of software vulnerabilities through regression testing. Therefore, whenever new code is committed to the repository, security testing should be performed against it. The naive way of accomplishing this is to execute all security tests on every commit. While this approach might be acceptable for a relatively small amount of test cases, it may not scale to the level of many test cases on a large software project with many commits occurring daily. To resolve this issue, testing of code needs to be prioritized. The first priority is to run tests against newly introduced code that has never been tested before. This can be thought of as testing for unit test compliance. The next priority is to test all code that the newly introduced code references, either directly or indirectly. It is possible that code that originally contained no detected bugs now has detectable bugs as the newly introduced code uses the older code in a way that was never done before. In this sense, this type of testing can be compared to integration testing.

Depending on the nature of the project being developed, as well as the software development background of the team members, it may be recommended, for some applications, to run SFCI as a standalone application, while for other applications, it may be best to integrate SFCI with a popular general purpose CI tool such as Jenkins. As an educational tool, users learning the very basics of software security can benefit from the default included tools in SFCI for detecting common security vulnerabilities by running it as a standalone application (Figure 3). For an external penetration testing firm, hired exclusively for detecting software vulnerabilities, using SFCI as a standalone tool can ensure that one proposed patch does not undo the efforts of a previously proposed patch. For more advanced projects, already using a CI server, SFCI can be integrated into the CI process and can provide important insight about the security status of their application to the software development team (Figure 4).

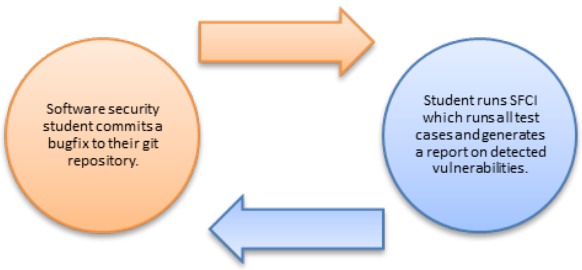

Figure 3. Single user using Security Focused Continuous Integration (SFCI) as a standalone application.

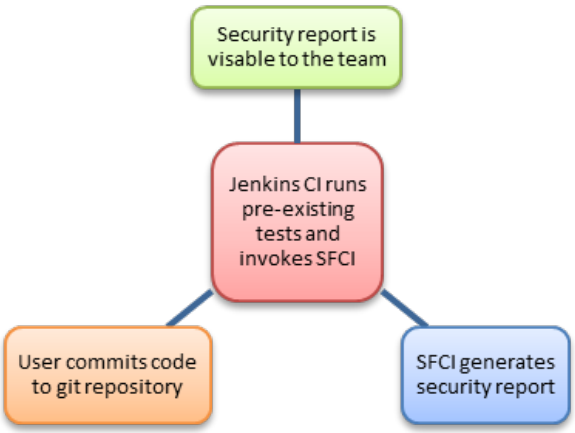

Figure 4. SFCI can be integrated with existing CI tools.

The remainder of this section of the paper will describe the recommended use of SFCI through two sample case studies.

6.1. SFCI as a Vulnerability Description Medium

While the goal of SFCI is to move the responsibility of software security closer to the main project developers, this does not imply that software penetration testers and security teams have been made obsolete. Indeed, all software developers should be educated on secure coding practices; and automated vulnerability detection tools should be used; however, despite this, there are still certain types of software vulnerabilities that can only be detected through "human cleverness". Therefore, the security experts for a software project should write test cases for software vulnerabilities that they have discovered and make those test cases executable under SFCI. The report generated by SFCI then serves as a communication medium [41] for describing the discovered vulnerabilities to the main project developers by providing them with information on the type of vulnerability, how to trigger it, and the location of the vulnerability within the project's source code.

6.2. SFCI for Local Security Testing

In order to prevent the security experts on a software project team from wasting their time discovering unsophisticated types of software vulnerabilities, it is best that these types of vulnerabilities be detected and corrected before they are added to any branch of the main project repository. To accomplish this, the security experts on the project team should create the SFCI test cases for types of vulnerabilities relevant to the project and distribute them to all developers. Each developer, then, should run these test cases against their code before they commit it to the main repository. If vulnerabilities are detected in their code, SFCI will alert them with easy-to-understand information about the type of vulnerability, its location, and how it can be triggered.

7. Testing an Example Project

A test was conducted against a web application we wrote with known vulnerabilities. This web application demonstrated all four types of input sanitization vulnerabilities checked for by our tool: SQL injection, command injection, HTML injection (XSS), and path traversal. In addition, the web application calls a program written in C, which can sometimes result in a use-after-free error. For developing our continuous integration tests, we followed the procedure described in our Getting Started Guide [40]. All our test cases for this example project are available on our GitHub page [42].

7.1. Developing the Test Cases

Testing for command injection vulnerabilities was straightforward as Commix was able to detect the command injection vulnerability in the value_x parameter and exploit it. The same command injection test case template that was used when developing this tool was used, and only the URL and

parameter tested by Commix needed to be changed. Testing for path traversal vulnerabilities was also straightforward as DotDotPwn was able to detect the path traversal vulnerability. The standard path traversal test case template was used, and only the base URL supplied to DotDotPwn needed to be changed. Testing for the use-after-free vulnerability was also straightforward as AddressSanitizer was able to identify the use-after-free whenever it occurred. The standard use-after-free template was used, and only the arguments passed to the program needed to be changed. Testing for HTML injection (XSS) vulnerabilities was more difficult as our original test case only used XSS Me's "Test all forms with all attacks" feature. This however, did not catch the XSS vulnerability present in our application as the vulnerability could only be triggered by an authenticated user. Our XSS test case needed to be modified so that it would only attempt to post XSS attack strings as an authenticated user. After this modification, our test case was able to detect the XSS vulnerability and properly report on it. Testing for SQL injection vulnerabilities was also difficult as sqlmap was unable to detect any vulnerabilities in our application. Instead, a test case was written to test an SQL injection vulnerability manually. Although the SQL injection vulnerability could not be automatically detected, this test case can still prove useful for detecting code regressions. If the bug is fixed, but then re-introduced, our tool will immediately alert the developers of this problem.

7.2. Evaluating the Test Cases

After placing all files in their appropriate directories, the run_LogParsing_image.sh script was executed. The JSON result files were then downloaded via netcat to the TestResults directory. The Python scripts, generate_pretty_src.py and render_website.py, were then executed. The main page, rendered/main.html, was then loaded in the Firefox web browser (Figure 5). Clicking on a vulnerability type displays more information about the vulnerabilities found in the project. For example, clicking on "Use-After-Free" displays information about the use-after-free vulnerabilities found in the project being tested (Figure 6). Clicking on a source code link on this page, either for the program source code itself or for a test case, will display a formatted (prism.js) version of this code, which will optionally display a line number in question (Figure 7).

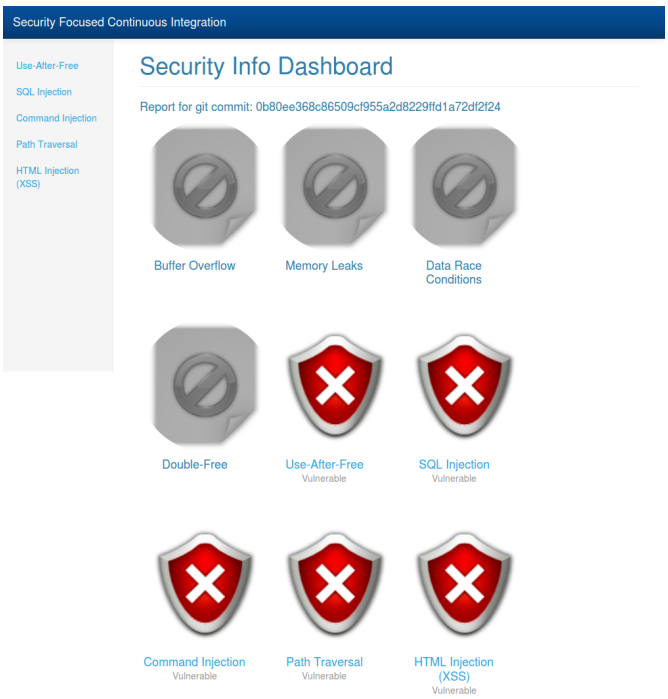

Figure 5. The main page displays the five categories of vulnerabilities that were found in the application. Gray icons correspond to vulnerabilities not tested for.

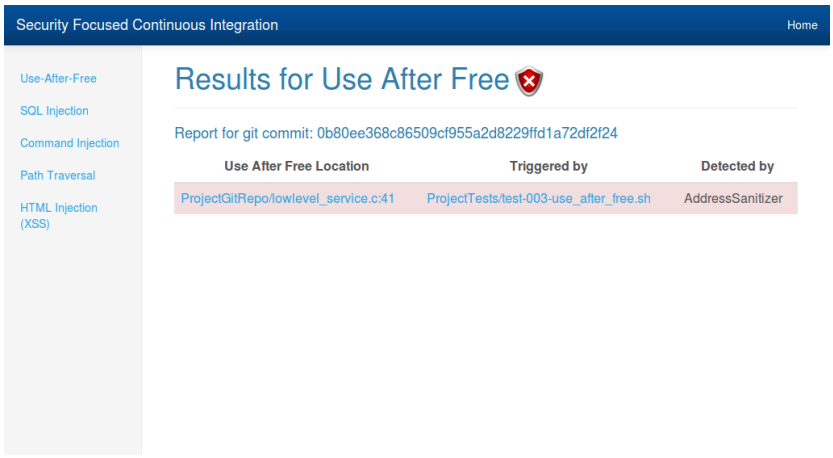

Figure 6. Report page for use-after-free vulnerabilities found in the project. Clicking on a link in the table of vulnerabilities will display the formatted contents of the file in question.

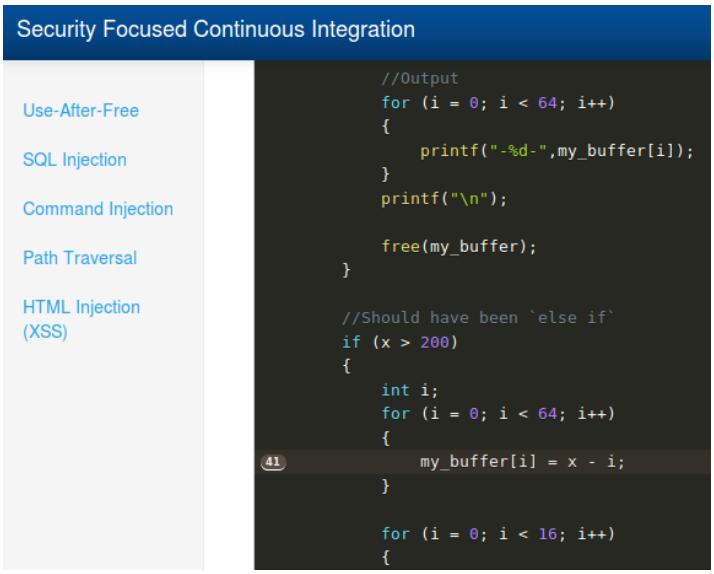

Figure 7. Clicking on a source code link displays the source code in question and an optional line number. This screenshot displays a use-after-free occurring on Line 41.

8. Testing ImageMagick

We tested our tool with a third party application, ImageMagick, which is a popular command line image processing package. ImageMagick has received much attention from the software security community due to the "ImageTragick" vulnerabilities. These vulnerabilities are the result of a lack of input sanitization, either in the form of insufficient shell character filtering allowing arbitrary command execution or only checking file extensions allowing an untrusted file containing Magick Vector Graphics (MVG) code to be read with full permissions as long as it has a trusted file extension such as .png. This latter described vulnerability allows an attacker to move, delete, and read arbitrary files [43].

8.1. Detecting a Known Vulnerability

Our tool was used to check for, and report on, a known vulnerability (CVE-2016-3715) in ImageMagick. This vulnerability exploited ImageMagick's "ephemeral" pseudo protocol and allowed an attacker to delete a file. Our test case consisted of using DotDotPwn to generate a list of path traversals and then check each one to see if it resulted in the deletion of a file. Successful paths were then listed in the web interface (Figure 8). When the insecure decoders were disabled by ImageMagick's

policy.xml file, which Debian's security update automatically did, no successful path traversals were found (Figure 9). The source code for these tests are available on our GitHub page [44].

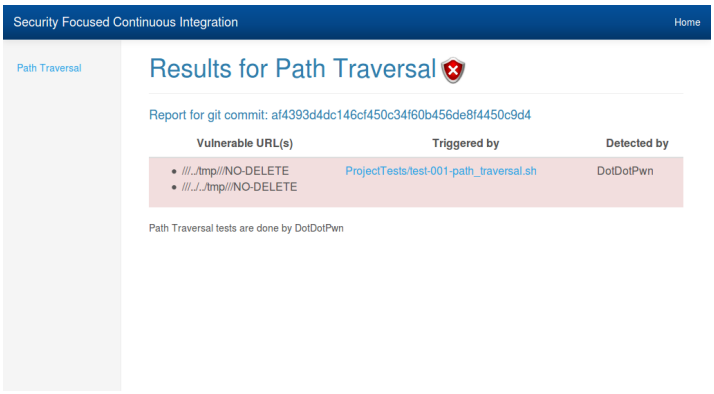

Figure 8. Results of the running test for CVE-2016-3715 with insecure decoders not disabled by ImageMagick's policy.xml.

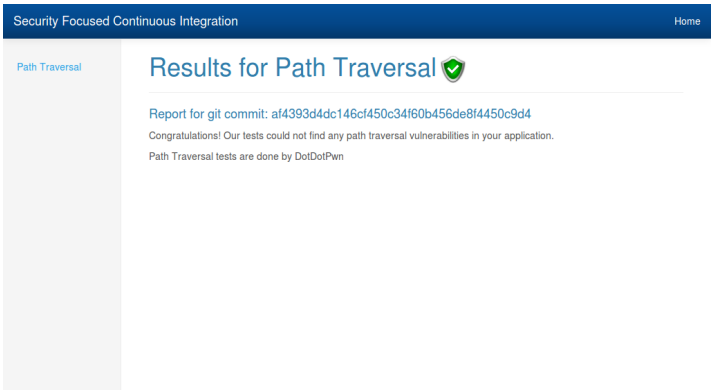

Figure 9. Results of the running test for CVE-2016-3715 with insecure decoders disabled by ImageMagick's policy.xml.

8.2. Detecting an Unknown Vulnerability

Our tool was used to check for, and report on, the possibility of an SVG file linking to any image file located on the target machine. DotDotPwn was used to generate a list of path traversals. Each trial consisted of embedding a link to the path traversal in an SVG file and attempting to rasterize the SVG file with the convert command. If the output file size was significant, greater than 1 KB, that file was examined to see if it contained the linked image. After a trial of Depth 8, no successful path traversals were found. Local SVG files are allowed, however, to link local image files from their same directory and any contained sub-folders [45]. ImageMagick will process an image according to the image type, which it identifies it as. This implies that if an SVG file, image.svg, is renamed to image.png, calling the ImageMagick command: convert image.png output.jpg will result in ImageMagick processing an SVG file, not PNG, and rendering it to a JPEG file. This introduces a potential information leakage vulnerability as the output JPEG file now could contain images from the local directory or its sub-directories, which were never meant to be part of the output. This vulnerability was reported to the ImageMagick developers who gave the recommendations of: disabling SVG decoders via the policy.xml file, executing ImageMagick within a secure environment such as a container, verifying images before they are passed to ImageMagick, or modifying the RegisterSVGImage() method to require explicit file names. Their response suggests that this vulnerability is more of a configuration vulnerability than an implementation vulnerability. In either case, our framework successfully demonstrated a test case for detecting what could be, in many use-case scenarios, an information leakage vulnerability.

9. Testing A Smart Grid Application

We now demonstrate how our tool can be used for finding vulnerabilities in software related to smart grids. The application that we tested is Open Energy Monitor's Emoncms web application [46]. This program has already received attention from the smart grid software security community, and several SQL injection bugs have been patched thanks to the researchers [47]. Our test case demonstrates checking for a Cross-Site-Request-Forgery (CSRF) vulnerability. This type of vulnerability occurs when the client-side web application communicates with the server-side web application using HTTP requests with easily guessable URLs. An attack website causes an unsuspecting user to click on a link that makes a request to the vulnerable URL on the server application. Any cookies associated with the legitimate web application are also sent, and therefore, the web application believes the request to be legitimate. In order to prevent this vulnerability, URLs need to be unique to the session. Unfortunately, this functionality is not implemented in Emoncms. The vulnerability arises because smart meters are allowed to use the HTTP API to push their data to Emoncms. Normally, this would require an API key (and therefore, a unique, hard-to-guess URL), but this key is not required when an administrator is logged in (cookies are used instead). Our test case opens Emoncms in the browser, logs-in as an administrator, then opens a new tab, and loads the attack site. The attack site has a link pointing to the following http://127.0.0.1/emoncms/feed/insert.json?id=3&time=UNIXTIME&value=8543.0, which when opened, wrongfully informs Emoncms that the user is using a large amount of power. Our test case considers CSRF vulnerabilities as XSS vulnerabilities as they share more characteristics in common than the other eight classes of vulnerabilities for which our application tests. After running the tests, the SFCI web interface displays the results listing the URLs vulnerable to CSRF (Figure 10).

Figure 10. The SFCI web interface displays the URL that has been found to be vulnerable to Cross-Site-Request-Forgery (CSRF).

10. SFCI Limitations

The scope of the SFCI tool is to provide software developers with constant feedback on the security status of their developed application through the use of pre-existing program analysis and penetration testing tools. Discovering new classes of vulnerabilities or improving the accuracy of detecting known classes of vulnerabilities lies within the scope of building penetration testing and dynamic analysis tools, but is outside the scope of the SFCI tool itself. In addition, the SFCI test cases must be manually written, and therefore, currently, the only automatic test case generation that can be used in a SFCI test case is that which comes pre-packaged in one of the included third-party tools. Automated SFCI test case generation, however, does lie within the scope of the SFCI tool and is noted as an item for future work. For example, it is possible to scan all source code files in a repository for code deemed to be potentially vulnerable, for example scripts that execute user supplied information as OS commands. Test cases could then automatically be generated that would use a tool such as Commix for testing for command injection vulnerabilities.

11. Conclusion and Future Work

As software continues to play an increasingly critical role in daily life, software developers need to be aware of the security issues associated with their software. A tool that creates automated reports on the security status of the program being developed whenever a programmer modifies its code can be instrumental in catching security bugs in the early stages of development, thus avoiding, or

minimizing the risk of, insecure software in production environments. This type of software quality control runs contrary to the silo model where application security is thought of as an isolated software component and proposes a different model where application security is continuously integrated throughout the entire development of the application. For future work, we plan to improve our continuous integration framework to support, by default, more dynamic analysis and penetration tools to be used in our test cases, especially for smart grid applications. In the future, we plan on giving SFCI the ability to generate test cases automatically, such as the example described in Section 10. Finally, our tool will offer the ability to report behaviours, if enabled, done by a deployed program, which may be indicative of an attack, such as submitted parameters containing SQL strings or common Unix commands, thus extending the continuous integration process to the deployment phase.

Author Contributions: Project administration, A.C.; Software, M.L.; Supervision, Q.H.M.; Writing—original draft, M.L.; Writing—review & editing, Q.H.M. and A.C.

Funding: This research was funded by Natural Sciences and Engineering Research Council of Canada grant number EGP 490684-15.

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

References

1. Rutkowska, J. The Three Approaches to Computer Security. Available online: http://theinvisiblethings.blogspot.ca/2008/09/three-approaches-to-computer-security.html (accessed on 20 September 2019).
2. Lindner, F.F. Software Security is Software Reliability. *Commun. ACM* **2006**, *49*, 57–61.
3. ThoughtWorks. Continuous Integration. Available online: https://www.thoughtworks.com/continuous-integration (accessed on 20 September 2019).
4. Jovanovic, N.; Kruegel, C.; Kirda, E. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In Proceedings of the 2006 IEEE Symposium on Security and Privacy, Berkeley/Oakland, CA, USA, 21–24 May 2006; pp. 258–263.
5. Sanchez, G. *Case Study: Critical Controls that Sony Should Have Implemented*; SANS Institute: Bethesda, MD, United States, 2015.
6. Yasar, H. An Introduction to Secure DevOps: Including Security in the Software Lifecycle. Available online: https://insights.sei.cmu.edu/devops/2016/11/an-introduction-to-secure-devops-including-security-in-the-software-lifecycle.html (accessed on 20 September 2019).
7. IriusRisk. BDD-Security. Available online: https://www.iriusrisk.com/open-source/ (accessed on 20 September 2019).
8. DeVries, S. Security Testing Embedded into Jenkins with BDD-Security. Available online: https://vimeo.com/89848072 (accessed on 20 September 2019).
9. DeVries, S. BDD-Security Wiki. Available online: https://github.com/continuumsecurity/bdd-security/wiki (accessed on 20 September 2019).
10. Ohlemacher, J. Valgrind Plugin. Available online: https://wiki.jenkins-ci.org/display/JENKINS/Valgrind+Plugin (accessed on 20 September 2019).
11. Adetoye, A. Zapper Plugin. Available online: https://wiki.jenkins-ci.org/display/JENKINS/Zapper+Plugin (accessed on 20 September 2019).
12. TinfoilSecurity. Simple DAST Overview. Available online: https://www.tinfoilsecurity.com/tour (accessed on 20 September 2019).
13. Irizarry, A. Tinfoil Security Plugin. Available online: https://wiki.jenkins-ci.org/display/JENKINS/Tinfoil+Security+Plugin (accessed on 20 September 2019).
14. Arachni. Checks. Available online: http://www.arachni-scanner.com/features/framework/#Checks (accessed on 20 September 2019).
15. Rohr, M. Automating DAST Scans with Jenkins, Arachni & ThreadFix. Available online: https://blog.secodis.com/2016/03/17/automated-security-tests-3-jenkins-arachni-threadfix/ (accessed on 20 September 2019).
16. GitLab. GitLab Continuous Integration. Available online: https://about.gitlab.com/gitlab-ci/(accessed on 20 September 2019).
17. GitLab. GitLab Runner. Available online: https://docs.gitlab.com/runner/ (accessed on 20 September 2019).

18. SG. Show Test Result with Web Interface (#18664). Available online: https://gitlab.com/gitlab-org/gitlab-ce/issues/18664 (accessed on 20 September 2019).
19. Berkers, G. How I run unit tests in Vagrant, in Jenkins. Available online: https://gielberkers.com/how-i-run-unit-tests-in-vagrant-in-jenkins/ (accessed on 20 September 2019).
20. Kuusela, J. Security testing in continuous integration processes. Available online: https://aaltodoc.aalto.fi/handle/123456789/27065 (accessed on 20 September 2019).
21. Lescisin, M.; Mahmoud, Q.H. Evaluation of Dynamic Analysis Tools for Software Security. *Int. J. Syst. Softw. Secur. Protect. (IJSSSP)* **2018**, *9*, 34–59.
22. Serebryany, K.; Bruening, D.; Potapenko, A.; Vyukov, D. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*; USENIX Association: Berkeley, CA, USA, 2012; p. 28.
23. ValgrindDevelopers. Projects Using Valgrind. Available online: http://valgrind.org/gallery/ (accessed on 20 September 2019).
24. OffensiveSecurity. Kali Linux Tools Listing. Available online: http://tools.kali.org/tools-listing (accessed on 20 September 2019).
25. Dalziel, H. World's Best 50 Firefox Pentesting AddOns. Available online: https://www.concise-courses.com/50-firefox-pentesting-add-ons/ (accessed on 20 September 2019).
26. Chacon, S.; Straub, B. A Short History of Git. Available online: https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git (accessed on 20 September 2019).
27. Ronacher, A. Welcome | Jinja2 (The Python Template Engine). Available online: http://jinja.pocoo.org/ (accessed on 20 September 2019).
28. QEMUDevelopers. KVM-QEMU. Available online: https://wiki.qemu.org/Features/KVM (accessed on 20 September 2019).
29. Gruhn, V.; Hannebauer, C.; John, C. Security of Public Continuous Integration Services. In Proceedings of the 9th International Symposium on Open Collaboration, Hong Kong, China, 5–7 August 2013; pp. 15:1–15:10.
30. TheClangTeam. AddressSanitizer-Clang 4.0 Documentation. Available online: http://clang.llvm.org/docs/AddressSanitizer.html (accessed on 20 September 2019).
31. ValgrindDevelopers. Valgrind Website. Available online: http://valgrind.org/ (accessed on 20 September 2019).
32. Stampar, M.; Damele, B. SQLMap Website. Available online: http://sqlmap.org/ (accessed on 20 September 2019).
33. ExploitDatabase. RealtyScript 4.0.2—Multiple Time-Based Blind SQL Injection. Available online: https://www.exploit-db.com/exploits/38497/ (accessed on 20 September 2019).
34. ExploitDatabase. WordPress Plugin GigPress 2.3.8—SQL Injection. Available online: https://www.exploit-db.com/exploits/37109/ (accessed on 20 September 2019).
35. Stasinopoulos, A. *Commix: Detecting and Exploiting Command Injection Flaws*; Blackhat Europe: London, UK, 2015.
36. SecurityCompass. XSSMe. Available online: https://github.com/SecurityCompass/XSSMe (accessed on 20 September 2019).
37. Navarrete, C.A.; Hernandez, C.C. dotdotpwn/README.md. Available online: https://github.com/wireghoul/dotdotpwn/blob/master/README.md (accessed on 20 September 2019).
38. ExploitDatabase. Home FTP Server 1.11.1.149—Authenticated Directory Traversal. Available online: https://www.exploit-db.com/exploits/15349/ (accessed on 20 September 2019).
39. ExploitDatabase. Femitter FTP Server 1.04—Directory Traversal. Available online: https://www.exploit-db.com/exploits/15445/ (accessed on 20 September 2019).
40. Lescisin, M.; Mahmoud, Q.H. Security Focused Continuous Integration (SFCI) Documentation—Getting Started Guide. Available online: https://github.com/uoitdnalab/SecureContinuousIntegration/blob/master/GettingStartedGuide.pdf (accessed on 20 September 2019).
41. Hilton, M.; Nelson, N.; Tunnell, T.; Marinov, D.; Dig, D. Trade-offs in Continuous Integration: Assurance, Security, and Flexibility. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 197–207.
42. Lescisin, M.; Mahmoud, Q.H. SecureContinuousIntegration/ExampleProject. Available online: https://github.com/uoitdnalab/SecureContinuousIntegration/tree/master/ExampleProject (accessed on 20 September 2019).
43. ImageTragick. ImageMagick Is On Fire - CVE-2016-3714. Available online: https://imagetragick.com (accessed on 20 September 2019).

44. Lescisin, M.; Mahmoud, Q.H. SecureContinuousIntegration/ImageMagick_FileDeletion. Available online: https://github.com/uoitdnalab/SecureContinuousIntegration/tree/master/ImageMagick_FileDeletion (accessed on 20 September 2019).
45. Heiderich, M. The Image that called me. In Proceedings of the OWASP Sweden Meeting, Gothenburg, Sweden, 14 April 2011.
46. OpenEnergyMonitor. Emoncms. Available online: https://emoncms.org/ (accessed on 20 September 2019).
47. Medeiros, I.; Neves, N.F.; Correia, M. Securing energy metering software with automatic source code correction. In Proceedings of the 11th IEEE International Conference on Industrial Informatics, INDIN 2013, Bochum, Germany, 29–31 July 2013; pp. 701–706.