FirmFuzz: Automated IoT Firmware Introspection and Analysis

Prashast Srivastava[†], Hui Peng[†], Jiahao Li[⋆], Hamed Okhravi[ξ], Howard Shrobe[⋆], Mathias Payer[‡]

[†]Purdue University, [⋆]MIT CSAIL, [ξ]MIT Lincoln Laboratory, [‡]EPFL

ABSTRACT

While the number of IoT devices grows at an exhilarating pace their security remains stagnant. Imposing secure coding standards across all vendors is infeasible. Testing individual devices allows an analyst to evaluate their security post deployment. Any discovered vulnerabilities can then be disclosed to the vendors in order to assist them in securing their products. The search for vulnerabilities should ideally be automated for efficiency and furthermore be device-independent for scalability.

We present *FirmFuzz*, an automated device-independent emulation and dynamic analysis framework for Linux-based firmware images. It employs a greybox-based generational fuzzing approach coupled with static analysis and system introspection to provide targeted and deterministic bug discovery within a firmware image.

We evaluate FirmFuzz by emulating and dynamically analyzing 32 images (from 27 unique devices) with a network accessible from the host performing the emulation. During testing, FirmFuzz discovered seven previously undisclosed vulnerabilities across six different devices: two IP cameras and four routers. So far, 4 CVE's have been assigned.

ACM Reference Format:
Prashast Srivastava[†], Hui Peng[†], Jiahao Li[⋆], Hamed Okhravi[ξ], Howard Shrobe[⋆], Mathias Payer[‡]. 2019. FirmFuzz: Automated IoT Firmware Introspection and Analysis. In *2nd Workshop on the Internet of Things Security and Privacy (IoT S&P'19), November 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3338507.3358616

1 INTRODUCTION

With 30 billion expected embedded devices by 2020 [7], the Internet of Things (IoT) has already proliferated across all aspects of our lives. The rise of IoT devices has been accompanied by increasing attacks on or through them. These attacks range from forming a botnet of embedded devices like the Mirai botnet [1] to a myriad of exploitable vulnerabilities that are reported in the corresponding firmwares [2, 3, 6, 8, 13].

In this work, we focus on analyzing Linux-based firmwares due to its widespread adoption. A Linux-based embedded firmware has three major parts that can be exploited by an adversary: (i) a variant of the Linux kernel, (ii) a set of open-source software packages, and (iii) a set of custom vendor-developed applications. The first two components are used in widely different contexts and can be vetted independently from embedded systems. The third component, custom vendor-developed applications, may be more prone to vulnerabilities because these proprietary applications are neither open-source nor openly vetted. We therefore focus on vendor-developed applications.

Evaluating vendor-developed applications in embedded firmware presents three challenges. First, these applications only accept syntactically legal inputs, therefore in order to perform deep analysis one has to infer and respect this syntax. Inferring the input syntax from a blackbox binary is a non-trivial problem [4, 15]. Second, to ensure triggered vulnerabilities do not escape the analysis, fine-grained monitoring of the runtime environment is required. Third, these applications rely on the runtime environment of the firmware for their functionality. Therefore, fuzzing these applications as standalone binaries is not sufficient.

In this paper, we describe an analysis framework, FirmFuzz, that finds *deep* vulnerabilities in vendor-developed applications of embedded firmwares. We overcome the above-mentioned challenges in FirmFuzz by: (i) utilizing the web application interface of these embedded devices as entry points to generate syntactically legal inputs, (ii) injecting runtime monitors into the runtime environment of the embedded firmware to allow for context-aware monitoring, and (iii) emulating the firmware image to keep our approach device-independent. To further enhance our greybox-based generational fuzz testing capability, we leverage information collected from static analysis to guide our fuzzer.

Previous efforts have studied emulation on a large-scale for closed-source firmware [2, 3]. These efforts focused on scaling an analysis to many images, but the undertaken analysis was generic, searching for specific vulnerabilities. An off-the-shelf analysis may miss vulnerabilities because it is not tailored to embedded systems and does not inspect the actions on the system.

Although we use web applications as an end-point for dynamic analysis, using web application scanners is insufficient. These scanners treat the analysis target as a blackbox

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

IoT S&P'19, November 15, 2019, London, United Kingdom
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6838-4/19/11...$15.00
https://doi.org/10.1145/3338507.3358616

while performing the vulnerability assessment. Emulating the firmware allows us to tune the runtime environment and to introspect the running system during execution, observing the vulnerability scanner interactions with the system. This, in turn, allows us to find *deep* vulnerabilities.

We provide a framework that, after some light-weight configuration, adapts to the emulated firmware and performs context-sensitive analysis. The focus of our work is not the *breadth* (number of images analyzed) but the *depth* (testing deep code paths for a variety of vulnerabilities) of the analysis undertaken.

Unlike conventional web application scanners, FirmFuzz leverages the degrees of freedom offered by an emulated firmware to enhance the vulnerability detection process. It integrates runtime monitors into the firmware filesystem, modifying the firmware itself to improve the bug finding process. Using our syntactically-legal input generation strategy and runtime monitors, we tailor our analysis on a per-firmware basis allowing us to trigger *deep* bugs in the firmware.

We analyzed 6,427 firmware images scraped from three vendors (Netgear, D-Link, and TRENDNet). Out of those, in 32, we found seven previously unknown vulnerabilities across six different devices comprising of two IP cameras and four routers. The vulnerabilities discovered include one post-authentication Command Injection (CI), three pre and one post-authentication Buffer Overflow (BO), one pre-authentication reflected XSS vulnerability and one pre-authentication Null Pointer Dereference (NPD). For the sake of responsible disclosure, we informed the vendors of these vulnerabilities.

In summary, we make the following contributions:

- We develop FirmFuzz (with open source code available at https://github.com/HexHive/FirmFuzz), an automated emulation and dynamic analysis framework for finding *deep* vulnerabilities in embedded firmware.
- We develop a generational fuzzer for syntactically legal input generation that leverages static analysis to aid fuzzing of the emulated firmware images while monitoring the firmware runtime (helper binaries and kernel monitors to enable deterministic bug discovery).
- We automatically test firmware images scraped from vendor websites and find seven previously unknown vulnerabilities.

2 FIRMWARE PREPROCESSING

FirmFuzz is a framework for the automatic analysis and fuzz testing of Linux-based IoT firmware through a QEMU-based emulation layer. It analyzes firmwares in three phases: information gathering, preparation, and fuzzing as depicted in Figure 1. Our framework currently supports analyzing MIPS-architecture and little endian ARM-architecture based firmware images.

2.1 Information Gathering Phase

This phase serves two goals. First, discovering the username-password pair required for authentication with the web application in order to increase the coverage of the fuzzer. Second, static analysis of the attack surface to find inputs for vulnerable code paths in PHP.

FirmFuzz brute-forces authentication credentials for a firmware through a crowdsourced credential corpus [11].

CI vulnerabilities in PHP-based applications arise from unsanitized user input being passed to unsafe PHP functions e.g., `system`, or `shell_exec`. FirmFuzz performs intraprocedural static taint analysis of the PHP scripts. It taints user-controlled variables, `$_GET`, `$_POST` and logs the code paths where the taint flows to unsafe functions. For each of these code paths, a constraint set is built. This set is used by FirmFuzz to generate inputs that can trigger vulnerable code paths.

2.2 Firmware Preparation

This phase creates a firmware image ready for emulation with a corresponding emulator configuration. A close approximate of the runtime environment as expected by the firmware is created using the peripheral mapping strategy. FirmFuzz leverages full-system emulation and injects helper binaries into the filesystem and augments the kernel to discover vulnerabilities with a low false positive/negative rate. The network of the emulator backend is configured to allow interaction between the firmware programs and the bug finding tools.

2.2.1 Peripheral Mapping. An embedded firmware often expects the presence of certain hardware peripherals during boot-time, runtime or both.

In the absence of these peripherals, a firmware may silently log an error or go into a busy loop trying to query the peripheral. In the former case, it is hard to infer what side-effect the absence of a peripheral may have on the functionality of the emulated firmware. In the latter case, the firmware may not reach a stable state.

If the firmware requests an unknown peripheral, FirmFuzz provides a mapping of that peripheral to a fake peripheral driver that always returns `True` on being queried. We acknowledge that this approach may not result in a stable state for all firmwares due to the diverse set of available IoT peripherals.

The firmware image is run under an emulator with our custom kernel. The kernel is configured to panic if unsupported devices are accessed. FirmFuzz uses this panic log to create a mapping of the device to the fake device driver. FirmFuzz iteratively performs this process until all the unsupported devices are mapped to a fake device.

2.2.2 Helper Injection. Helper binaries allow FirmFuzz to inspect the firmware during emulation. The helpers operate within the runtime environment of the firmware and in the current implementation allow us to detect CI vulnerabilities. During fuzzing, if this helper binary is executed, FirmFuzz flags a CI vulnerability. This approach is completely automated and firmware agnostic; i.e., we are not reliant on the utilities present on the firmware to detect the vulnerability. The closest previous work to ours, Firmadyne [2], does not have support for the detection of CI vulnerabilities. The

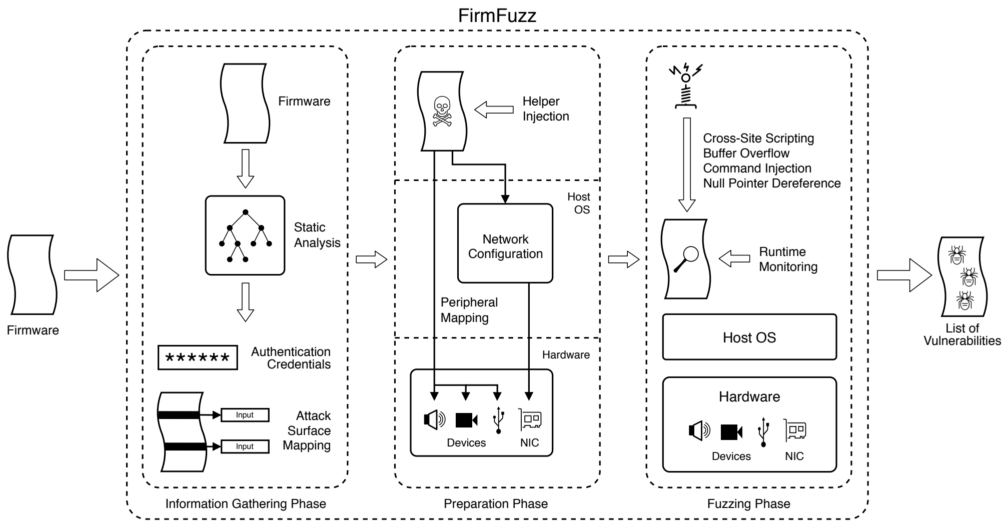


Figure 1: FirmFuzz Workflow

ones reported in its paper were discovered through manual analysis of the webpages.

To detect BO vulnerabilities, we utilize the exception handling mechanism of the Linux kernel similar to Firmadyne. FirmFuzz leverages the mechanism further to detect NPD vulnerabilities. The major difference between FirmFuzz and Firmadyne in detecting BO is that the Firmadyne authors manually discovered a BO vulnerability in a webpage served by a specific Netgear router which they validated by crafting a curl request to that specific webpage to trigger that vulnerability. However, FirmFuzz requires no manual analysis to detect BO vulnerabilities. Using the custom fuzzing driver, it automatically triggers BO vulnerabilities in the emulated firmware and packages a PoC exploit to recreate the vulnerability. For the XSS vulnerability targeted by FirmFuzz, we observed that host-side monitoring is sufficient for detection.

2.2.3 Network Configuration. Firmware images differ in how they name and assign addresses to their LAN/WAN interfaces. We follow the same approach as Firmadyne to infer these network configurations. We first run the emulation in a 'network inference' mode in which FirmFuzz logs all the interaction of the firmware with the networking interface of the kernel. Using these logs, FirmFuzz infers the network configuration and creates the appropriate virtual network interfaces to allow interaction with the emulated firmware.

3 FIRMWARE FUZZING

FirmFuzz detects vulnerabilities using a custom-developed automated generational fuzzer. Existing vulnerability scanning approaches for embedded firmware [2, 3, 16] require human guidance.

FirmFuzz changes the paradigm of vulnerability detection in emulated firmware by augmenting the emulation environment of the fuzzed target to aid in vulnerability discovery,

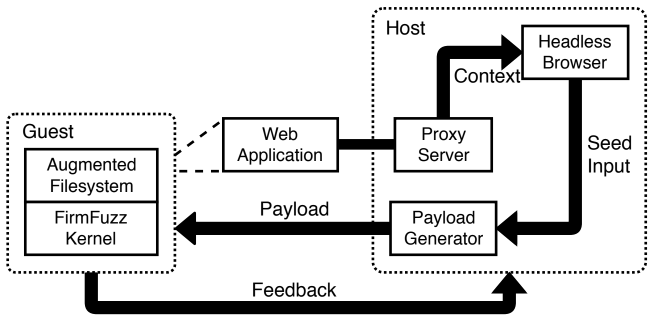


Figure 2: Fuzzing Workflow

see Figure 2. Our approach removes the reliance on server feedback and allows a direct observation of the triggered vulnerability *in situ.* Three main features of the FirmFuzz fuzzer are: (i) *Context-driven input generation* — It incorporates contextual information provided by the firmware while interacting with different parts of the attack surface, (ii) *Deterministic vulnerability detection* — The vulnerability monitors operating both in the guest (i.e., the emulated firmware) and the host allow deterministic vulnerability detection, and (iii) "Fuzzing side-effects" elimination — FirmFuzz with the help of its emulation framework, automatically reverts the firmware back to a stable state if the firmware reaches an inconsistent state while being fuzzed. This allows continuous fuzzing of the emulated target without requiring manual intervention to reset the firmware state.

The web application setup by the firmware provides its functionality by employing a combination of client-side JavaScript code and server-side code. FirmFuzz uses a headless browser controlled by our fuzzer to interact with the firmware through the web application to execute the client-side JavaScript code. Using the contextual information from these interactions, it generates fuzzing inputs. All network traffic between the fuzzer and the emulated firmware passes through a proxy server. This allows the proxy server to capture candidate inputs that will be mutated by FirmFuzz.

The four vulnerability types targeted by FirmFuzz during the fuzzing phase are: CI, BO, NPD, and XSS. To detect these vulnerabilities, FirmFuzz first interacts with the web application to generate a legal HTTP request as seed input to fuzz a part of the attack surface, see Section 3.1. It then mutates the legal request with payloads based on the vulnerability being targeted, and sends it to the firmware. Upon sending a mutated request, FirmFuzz monitors the firmware to detect vulnerabilities. If a vulnerability is detected, the mutated request is logged as a Proof-of-Concept (PoC) input along with the recipient URL to allow reproducibility.

3.1 Syntactically Legal Input Generation

Since FirmFuzz uses a headless browser for firmware interaction, the burden of creating a well-formed HTTP request that effectively tests the vendor-written software is offloaded to the web application. Additionally, the browser handles executing client-side code to give itself access to the full functionality of the application.

However, using a web application interface as an oracle to generate syntactically-legal input opens a challenge. FirmFuzz must be aware of the web interface setup of the emulated firmware in order to interact with it successfully. During our experiments, we observed that the web interface setup highly varies across vendors and devices. During evaluation of FirmFuzz, based on the images we fuzzed, we created templates for interacting with the encountered web interfaces. Note, however, that this interface support requires some manual validation by the analyst. This minimal manual validation is the limiting factor for scaling the analysis to additional devices. This highlights one of the primary differences between FirmFuzz and the previous work in this area: we strive to perform deeper analysis, but at a *slight* cost to scale.

3.2 Deterministic Bug Discovery

Current *automated* approaches for detecting vulnerabilities (CI, BO and NPD) employed by scanners like ZAP [16] are reliant on the server-side response for detecting them. This approach is imprecise as it may either fail to localize a vulnerability, e.g., BO, NPD, or miss a vulnerability altogether, e.g., CI if a particular Linux utility does not exist. Even if a vulnerability is detected, there is no guarantee that it exists since the server-side response *may* be inaccurate.

To deterministically detect the above mentioned vulnerabilities, FirmFuzz modifies the emulation environment of the firmware and the firmware itself.

FirmFuzz detects CI, BO and NPD vulnerabilities by monitoring the logs generated by the augmented firmware when a test input is executed. For CI, it monitors the `execve` system call to log execution of the poison binaries (see Section 2.2.2). For BO and NPD, it monitors the kernel logs to see if any firmware process tried to access unmapped memory.

Detecting an XSS vulnerability does not require any guest-side assistance; host-side monitoring by FirmFuzz is sufficient to detect it, similar to existing vulnerability scanners. This monitoring is in the form of detecting if the snippet of JavaScript code that was sent as a payload in the request is executed in the machine.

3.3 Elimination of Fuzzing Side-Effects

Since FirmFuzz actively interacts with the emulated firmware during fuzzing, it may become unresponsive. The inconsistent states include the firmware trying to reboot itself to apply certain changes or an infinite cycle due to the triggered functionality being emulated incorrectly. In such a state, the firmware becomes non-responsive to fuzzing and requires a roll back to a stable state.

The above problem often requires manual intervention to perform the rollback. However, to keep the fuzzing process fully automated, FirmFuzz employs a "snapshot and rollback" strategy. During the dynamic analysis, if the firmware is pushed into an inconsistent state, FirmFuzz reverts the emulated firmware back to a stable state (right after the completion of initialization) to be ready for the next request. This allows for continuous fuzzing of the target without manual intervention.

3.4 Payload Delivery

The web application performs client-side validation checks on the input values, such as checking if the input values are formatted according to the syntax expected by the application. Therefore, to effectively fuzz the firmware, a legal request from the web application is required, which can be mutated and sent to the firmware directly bypassing the client-side validation checks.

Our fuzzer requires a set of seed inputs and valid URLs that are accepted by the web application. As web applications often consist of a JavaScript host component and a server component, we need to trigger the generation of all possible host paths by driving their host component. FirmFuzz leverages the headless browser to load webpages from the firmware and then walks through the DOM to trigger all possible state changes by iterating through all button elements on the web page (often firing JavaScript events along the way). In addition, our host components fill input fields with syntactically *legal* values that are inferred through the names of fields and a set of possible data ranges.

If FirmFuzz infers legal input values, then a seed request is generated by interacting with a button element. This seed request is mutated for fuzzing. The mutation strategy followed by FirmFuzz is a substitution strategy in which the relevant parameters of a request are modified to contain malicious payloads. This substitution strategy is used across all the vulnerability detection modules of FirmFuzz. The substituted payload is adjusted based on the vulnerability being tested. Note that a finite number of payloads are tested for each vulnerability. Therefore, a fixed number of mutants are created for each seed request. A more detailed overview of this process is provided in Appendix A

4 EVALUATION

We evaluate FirmFuzz by testing it on a set of 6,427 firmware images. We first give a breakdown of the dataset on which FirmFuzz was able to be evaluated as well as the number of unique web interfaces encountered. We discuss one of our discovered bugs as a case study and how the discovered vulnerabilities are not detected by existing state-of-the-art tools. We also evaluate our runtime performance and discuss our vulnerability detection accuracy.

FirmFuzz is evaluated on a machine with an Intel i7 processor and 16GB RAM and running Ubuntu 16.04. To emulate firmwares, FirmFuzz uses QEMU [9] version 2.5.0 as the emulation backend. For fuzzing, FirmFuzz uses a headless browser as one of its fuzzing drivers. The headless browser used is the Selenium WebDriver [12] version 3.4.0. In addition, FirmFuzz uses a proxy server, mitmproxy version 0.18.2 [5], to monitor the network traffic sent between the fuzzer and the emulated firmware.

4.1 Firmware Images Tested

6,427 images were scraped from three vendor websites to create our image dataset. From the dataset, 1,013 images had a Linux-based File System (FS). Out of these, 203 images had their network configuration inferred and 32 images from this set had accessible web interfaces which were used by FirmFuzz as entry points for fuzzing the image. The breakdown is presented in Table 1.

There is a sharp drop-off between the images for which the network configuration was inferred and those which were successfully fuzzed. This drop-off occurs due to missing emulation for specific required devices (e.g., a system waits for a camera to be accessible before starting the web server). Without an accessible web interface as an entry point, FirmFuzz cannot fuzz the firmware image.

Out of the fuzzed images,there is high reusability of web interfaces between different devices from the same vendor — 6 unique web interfaces in 27 unique device images across 3 different vendors. Therefore, with minimal manual effort we can cover a large number of emulated devices.

4.2 Case Study

FirmFuzz, using its analysis detects vulnerabilities hidden deep in the firmware which are not immediately apparent. A case study of such a vulnerability is presented below.

4.2.1 *TRENDnet TEW-673GRU Router.*
This is a MIPS-based Wireless Gigbabit router. A CI vulnerability was found in this wireless router. It can be remotely exploited if a user is logged in to the device's configuration webserver with administrative credentials.

The vulnerability exists in the vendor-written program `timer` on the firmware. This program runs by default as a daemon on the router with root privileges. The `timer` uses another vendor-written software `arpping` to periodically check if the router is reachable every three minutes.

The vulnerability lies in the passing of the parameters from `timer` to `arpping`. Five parameters are retrieved from device memory and passed to `arpping` without any validation or sanitation. Out of the five parameters, three of them are under user-control through a web application setup by the router which performs client-side validation on these parameters.

FirmFuzz using its syntactically legal input generation, inferred the input request and the CGI binary responsible for updating those user-controlled parameters. FirmFuzz then sent a `POST` request targeted at discovering a CI vulnerability directly to the CGI binary which updates the device memory with the sent values respectively without validating them. Using the firmware runtime monitoring, FirmFuzz detected the vulnerability because it was triggered every three minutes by the `timer` program.

4.3 Comparison with Existing Analysis Frameworks

To show the effectiveness of FirmFuzz, we compare against other state-of-the-art open-source web vulnerability scanners. We chose two of the most popular ones, Zed Attack Proxy (ZAP) [16] and w3af [14], for our evaluation. We also evaluated the automated vulnerability detection of Firmadyne on our sample set of firmware images as well. The evaluation was performed on the basis of the number of vulnerabilities found by the tools.

4.3.1 *Firmadyne.*
Firmadyne runs a set of Metasploit modules of known exploits for embedded devices during its automated dynamic analysis. Additionally, it tests the emulated image for a set of vulnerabilities that the authors of the framework found manually in some embedded devices.

As evident from Table 2, this approach, even though completely automated and applicable at large-scale, fails to find any of the vulnerabilities in our sample set of firmware images. The Firmadyne tests were originally built for specific embedded devices and the probability of the same exploit working across the different devices and different vendors is low.

4.3.2 *w3af.*
As w3af cannot infer the credentials to the administrative interface by itself, we provide it as a head start for a fairer comparison. However, even after providing the necessary credentials to the authentication plugin and configuring several parameters including the HTML tags for the input fields manually, w3af was still unable to authenticate with the firmware. This is the reason why w3af failed to detect any of the vulnerabilities detected by FirmFuzz.

4.3.3 *Zed Attack Proxy.*
To ensure that ZAP had access to the same attack surface as FirmFuzz, the credentials to the firmware for administrative access were provided to ZAP as it does not have the authentication discovery capability of FirmFuzz. With the credentials provided, we ran the *Active Scan* feature of ZAP on the emulated image which is an

Table 1: **Firmware images tested**

Vendor	Scraped Images	Linux FS found	Network Inferred	Fuzzed (Unique Devices)	Unique Web UI
TRENDnet	359	129	26	6 (5)	2
Netgear	2646	675	162	20 (17)	3
D-Link	3422	209	15	6 (5)	1
Total	6,427	1,013	203	32 (27)	6

Table 2: Vulnerability Detection Comparison of FirmFuzz against Zed Attack Proxy(ZAP), Firmadyne and w3af

Number	Vulnerability	Vendor	Device	CVE-ID	FirmFuzz	ZAP	Firmadyne	w3af
1	Command Injection	TRENDnet	TEW-673GRU	CVE-2018-19239	✓	✗	✗	✗
2	Reflected XSS	TRENDnet	TEW-634GRU, 673GRU, 632BRP,	–	✓	✓	✗	✗
3	Buffer Overflow	TRENDnet	TEW-673GRU, 632BRP	CVE-2018-19242	✓	✗	✗	✗
4	Buffer Overflow	TRENDnet	TV-IP110WN, IP121WN	–	✓	✗	✗	✗
5	Buffer Overflow	TRENDnet	TV-IP110WN, IP121WN	CVE-2018-19240	✓	✗	✗	✗
6	Buffer Overflow	TRENDnet	TV-IP110WN,IP121WN	CVE-2018-19241	✓	✗	✗	✗
7	Null Pointer Dereference	Netgear	DG834	–	✓	✗	✗	✗

automated scan feature that tries to find vulnerabilities by deploying known attacks including the ones FirmFuzz targets.

As can be observed in Table 2, ZAP only discovered the XSS vulnerability but failed to discover any other vulnerability. This is because ZAP treats the firmware as a black box during its automated scan and does not actively interact with the firmware to gain context about the application like FirmFuzz does. Additionally, ZAP does not have the capability to monitor the runtime environment of the firmware to detect any erroneous conditions while performing the scan. This limits its capability to detect those vulnerabilities which, when triggered, do not provide any feedback to the scanner through the web application entry point.

Even if the PoC for the discovered vulnerabilities in the image are provided to ZAP, it still cannot infer their existence. This is because ZAP relies on overtly observable signals for vulnerability detection. For example, for CI vulnerabilities, ZAP relies on server-side replies. The vulnerabilities we discovered do not send such a response from the server when triggered because they exist in an auxiliary program that does not interact with the web application directly.

4.4 Runtime Performance

We further evaluate the runtime performance of FirmFuzz during the fuzzing phase. The average runtime for the fuzzing phase is *16m 42s*. The comparatively low runtime overhead is primarily because our fuzzer is a generational fuzzer rather than a mutational one. With our generational fuzzer, we constrain the state space of inputs drastically, thus achieving better overhead.

4.5 Vulnerability Detection Accuracy

As shown in Table 2, existing automated scanners incur a high false negative (FN) rate when testing web applications. Their automated scans employ a brute-force approach to detect vulnerabilities. The brute-force approach includes strategies such as mutating the URL under test with payloads and performing blind directory traversals. The brute-force approach, reliance on server feedback for detecting such vulnerabilities, and the the automated scans being blind, i.e., they do not actively interact with the application to gain context about the web application leads to the high FN rate.

FirmFuzz using its runtime monitoring of the firmware under test along with the contextual input generation lowers the number of FN bugs while detecting BO, NPD, and CI compared to the existing automated scanners. We do not completely remove all instances of FN bugs since FirmFuzz relies on template request generation for fuzzing. Therefore, if a request is not generated for a particular page using our heuristics then a bug may be missed. However, such instances are clearly logged by FirmFuzz and an analyst can provide feedback to FirmFuzz in terms of acceptable input for the page which can be used to lower the chances of a FN bug.

As discussed in Section 2.2.2, FirmFuzz monitors the execution of a poison binary and memory access violation handler in the kernel for detecting CI and BO, NPD respectively. These detection methods not only ensured that none of the bugs caught by FirmFuzz that belonged to the class of CI, BO, and NPD were false positives (FP) but also gave information which firmware resource was buggy. This is a drastic improvement over the existing scanners which, due to their lack of system introspection, are neither able to localize a bug if it exists or give guarantees that the bug detected by these scanners is not a FP warranting further manual analysis.

5 RELATED WORK

A large body of work has contributed to security analysis of firmware images for embedded devices. However, the closest efforts to our work in terms of the targeted device domain (i.e., embedded Linux-based devices) are the framework by Costin et al. [3] and Firmadyne, the framework by Chen et al. [2].

5.1 Firmadyne

Chen et al. [2] presented Firmadyne, a full-system emulation based framework for dynamic analysis of embedded firmwares. They carried out the most extensive analysis in terms of the number of firmware images analyzed to date. However, dynamic analysis done by Firmadyne was simple. Their only automated vulnerability discovery pass consisted of running known exploits as Metasploit modules and their own PoC for manually discovered vulnerabilities.

Running a pre-defined set of exploits, while helpful in finding known vulnerabilities, is not effective in discovering new ones (as evident from Table 2) since the probability of the same exploit invoking different vulnerabilities across different classes of devices and vendors is low. On the contrary, FirmFuzz tailors the payloads to the target emulated firmware allowing it to test deep code paths and finds new vulnerabilities.

5.2 Firmware Analysis by Costin et al.

Costin et al. [3] built an emulation framework targeted specifically at emulating the web interface of the firmware and performed static and dynamic analysis on it.

Their approach was dependent on existing tools, RIPS [10] for their static analysis of PHP scripts, vulnerability scanners like [16] for the dynamic analysis. These tools, however, have high chances of false positive and false negative rates respectively.

FirmFuzz, on the other hand, incorporates its own custom tools into the framework to lower the false positive/negative rates. The static discovery module is constrained to look only for potential CI vulnerabilities and is designed to output constraints for each vulnerable code path present. The dynamic analysis module (i.e., the fuzzer) is aware of the other modules in FirmFuzz and leverages all information available from them, e.g., output from the runtime monitors on the emulated image and the information acquired during the static analysis phase. This provides enhanced bug discovery.

6 CONCLUSION

The increasing range of IoT devices have access to our personal data and impact our everyday life, calling for additional scrutiny when evaluating their security. We present FirmFuzz, an automated framework for whole-system emulation and fuzz testing of embedded firmware images. We used FirmFuzz to test for four types of vulnerabilities in the firmware images that we studied: CI, BO, NPD and XSS. We found and reported seven previously undiscovered vulnerabilities using FirmFuzz.

ACKNOWLEDGMENTS

This research was supported by ONR award N00014-17-1-2513 and by NSF CNS-1801601. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Manos Antonakakis, Tim April, Michael Bailey, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, Yi Zhou, Manos Antonakakis Tim April Michael Bailey Matthew Bernhard Elie Bursztein, Bullet J Jaime Cochran Zakir Durumeric Alex Halderman Luca Invernizzi, Bullet Michalis Kallitsis Deepak Kumar Chaz Lever Zane Ma, Joshua Mason Damian Menscher, Bullet Chad Seaman Nick Sullivan Kurt Thomas, and Bullet Yi Zhou. 2017. Understanding the Mirai Botnet. In *Proceedings of the 26th USENIX Security Symposium*. 1093–1110. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[2] Daming Dominic Chen, Manuel Egele, Maverick Woo, and David Brumley. 2016. Towards Fully Automated Dynamic Analysis for Embedded Firmware. *Network and Distributed System Security Symposium* February (2016), 21–24. https://doi.org/10.14722/ndss.2016.23415

[3] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2016. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. *Proceedings of the 2016 ACM Asia Conference on Computer and Communications Security (AsiaCCS'16)* (2016), 437–448. https://doi.org/10.1145/2897845.2897900 arXiv:1511.03609

[4] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine learning for input fuzzing. In *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 50–59. https://doi.org/10.1109/ASE.2017.8115618 arXiv:1701.07232

[5] Mitmproxy. 2018. mitmproxy. https://mitmproxy.org/.

[6] Antoine Nervaux. 2018. Vulnerability disclosure TP-Link multiples CVEs - pentest - try harder. https://chmod750.com/2017/04/23/vulnerability-disclosure-tp-link/.

[7] Amy Nordrum. 2018. Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated - IEEE Spectrum. https://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated.

[8] Kim Pierre. 2018. Pwning the Dlink 850L routers and abusing the MyDlink Cloud protocol - IT Security Research by Pierre. https://pierrekim.github.io/blog/2017-09-08-dlink-850l-mydlink-cloud-0days-vulnerabilities.html.

[9] QEMU. 2018-01-08. QEMU. https://www.qemu.org/.

[10] RIPS. 2018. RIPS - PHP Static Analyser. https://www.ripstech.com/.

[11] Router. 2018. Default Router Passwords - The internets most comprehensive router password database. http://routerpasswords.com/.

[12] Selenium. 2018. Selenium - Web Browser Automation. http://www.seleniumhq.org/.

[13] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. *Proceedings 2015 Network and Distributed System Security Symposium* February (2015), 8–11. https://doi.org/10.14722/ndss.2015.23294

[14] w3af. 2018. w3af: web application attack and audit framework. http://w3af.org/.

[15] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *Proceedings - IEEE Symposium on Security and Privacy*. 579–594. https://doi.org/10.1109/SP.2017.23

[16] ZAP. 2018. OWASP Zed Attack Proxy Project - OWASP. https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

A PAYLOAD DELIVERY

In this section, we provide an extended description of how FirmFuzz generates inputs to fuzz test the emulated firmware. FirmFuzz uses Algorithm 1. First, on receiving a webpage served by the web application, FirmFuzz finds all the button elements in it. These elements are used to interact with the firmware and are added to a list using the `find_buttons` subroutine.

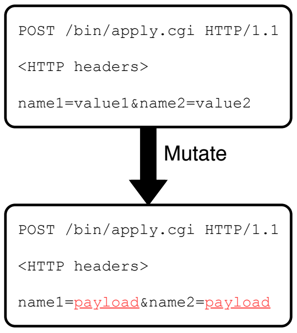
Algorithm 1 Payload delivery

```
Require:
  WebPage: The current webpage being evaluated
  Firmware: The emulated firmware
  procedure DELIVER_PAYLOAD(WebPage)
      buttons[] ← $WebPage.find\_buttons()$
      input[] ← $WebPage.infer\_input()$
      while len(buttons)>0 do
          send_mutate(input, buttons[0])
          if Firmware.isInconsistent() then
              Emulation.restore( )
          end if
          buttons.pop( )
      end while
  end procedure
```

Second, FirmFuzz needs to infer legal values for the input fields presented by the interface performed by the `infer_input` subroutine. FirmFuzz employs two strategies based on whether an input field is empty or non-empty: (i) *Non-empty field*—Field is populated with a default value filled in by the firmware. In this case, FirmFuzz leaves the value unchanged. (ii) *Empty field*—FirmFuzz does a pattern match on the HTML ID attribute of the corresponding input element. Based on whether the ID attribute contains familiar strings such as "mac" or "IP", a dummy MAC address or an IP address is filled into the input field respectively. If none of these strings match then FirmFuzz fills the input field with dummy text.

The names and types of these ID elements (parameters) are not based on a standard but chosen by vendors who developed the web application. Therefore, it is possible that the heuristic employed by FirmFuzz unable to infer the legal value requested.

In the event that FirmFuzz incorrectly infers values for the input fields and cannot make the web application generate a request, the "ID" attribute is logged for manual classification later as either an input "ID" for a MAC or an IP address. FirmFuzz maintains a database for the incorrectly inferred input elements so that future runs of FirmFuzz can better infer legal values.



```
POST /bin/apply.cgi HTTP/1.1
<HTTP headers>
name1=value1&name2=value2
```

```
POST /bin/apply.cgi HTTP/1.1
<HTTP headers>
name1=payload&name2=payload
```

Figure 3: FirmFuzz Mutation Strategy

If FirmFuzz manages to infer legal input values, then `send_mutate` subroutine will generate a seed request by interacting with a button. This seed request will then be mutated for fuzzing. The mutation strategy followed by FirmFuzz is a simple substitution strategy in which the relevant parameters of a request are modified to contain malicious payloads as depicted in Figure 3. A similar strategy is followed for `GET` requests to mutate its parameters. This substitution strategy is used across all the vulnerability detection modules of FirmFuzz. The substituted payload is adjusted based on the vulnerability that is being tested by FirmFuzz.

At any point during the seed request generation or the mutated requests delivery, the firmware reaches an inconsistent state, the emulation is rolled back automatically to a stable state by FirmFuzz and the fuzzing is carried forward.