Tool development for practical embedded software analysis and vulnerability discovery

Denis Vasilyev

**Introduction**

The prevalence of Internet of Things (IoT) devices in our daily lives is rapidly increasing. It's predicted that the number of connected IoT devices will surge to 14.4 billion by the end of 2023, and over 25 billion by 2030. However, unlike general-purpose computers, IoT devices are much more limited in terms of hardware and processing power, rendering traditional security measures impractical. This makes them highly susceptible to attacks and malicious exploitation, with the potential for compromised devices to be used in launching cyber-attacks. An infamous instance of this was the Mirai botnet, a malware that affected IoT devices running on the ARC processor. By exploiting default usernames and passwords, Mirai is able to infect devices and allow attackers to send spam, crack passwords, and initiate DDoS attacks.

**Vulnerability Discovery**

Most commonly, vulnerabilities in embedded systems occur due to undiscovered or unfixed bugs in the firmware. This can happen due to a number of reasons, such as poor coding practices, reusing old code that already has those vulnerabilities or human error. Embedded devices are generally more prone to vulnerabilities due to limitations in power and the emphasis on performance in such systems. Additionally, there are some language limitations that often require additional safeguards to be implemented to avoid causing vulnerabilities, especially in languages with manual memory management like C, which is commonly used in embedded software due to its high performance. Discovering those vulnerabilities can be a challenge for the developers’ due limitations in time and budget, as well as due to the high level of cybersecurity knowledge required to spot those vulnerabilities. Finding those vulnerabilities is even more difficult for customers and users of the IoT device, due to generally not having access to the source code of the firmware. Nonetheless, especially in a defence context, it is crucial that the vulnerabilities can be quickly and systematically discovered and patched, creating a need for tools to assist in the discovery of such vulnerabilities.

There are two main categories of techniques that can be used to find vulnerabilities in firmware, static analysis, and dynamic analysis. Ideally both static and dynamic analysis should be used when attempting to find vulnerabilities, as dynamic analysis could be used to test a potential vulnerability found through static analysis to ensure that it does actually exist and hasn’t been fixed by an obscure piece of code. Static analysis is a technique that aims to detect vulnerabilities by analysing the code itself and matching patterns to known vulnerabilities and unsafe code techniques. A major challenge in using static analysis on embedded IoT devices is the lack of access to source code in the vast majority of cases. A decompiler such as Ghidra can bypass this issue somewhat, potentially assisted by several Ghidra scripts to assist in the analysis of decompiled code and pattern matching to known vulnerabilities, however the decompiled code is often incomplete or obscured. This gets even more difficult if optimisation is used when compiling the source code, which is often utilised when creating embedded firmware due to optimised code minimising the impact of the hardware limitations of the device. Other methods of static analysis can include extracting firmware using binwalk[[1]](#footnote-1), and then using Linux commands like grep to find poorly protected default passwords, as well as using other tools like OWASP to statically analyse binaries.

On the other hand, dynamic analysis is a technique that involves running the firmware in a real environment and performing manual or automatic testing to find bugs and vulnerabilities in the firmware. There are a number of tools that can be used for automatic testing, including fuzzers such as AFL[[2]](#footnote-2), as well as penetration testing software like Metasploit[[3]](#footnote-3). Fuzzing involves providing invalid or malformed inputs into a program, aiming to trigger a vulnerability such as a stack overflow, or to inject a malicious command into the device. During this project, both static and dynamic analysis were investigated, but the main focus was on dynamic analysis tools and techniques. When performing dynamic analysis, ideally penetration testing software would be run on a physical device, however due to the nature of embedded IoT devices this is generally not possible or impractical, as the hardware limitations of such devices would greatly reduce the effectiveness of such methods. Additionally, such a method is not scalable due to the cost of purchasing each device that needs to be tested, making large scale vulnerability discovery impractical. Additionally, in a defence context, testing a physical embedded system could require a larger system in which the targeted device is integrated into to be non-operational during testing.

**Analysing the Archer A7**

One of the initial goals of this project was to look into and develop tools to analyse the Archer A7 router firmware. The specific version of the firmware is 210519, a version that has a publicly known vulnerability in its tddp binary[[4]](#footnote-4). Initially, the firmware was extracted and decompiled using Ghidra, and after some searching, the vulnerability was found. However, it would be very difficult to find such vulnerability if it wasn’t already known that it was there, so other methods of vulnerability discovery were investigated, with focus on dynamic analysis. The end goal was to be able to systematically find such vulnerabilities in a router like the Archer A7. Since there was no access to a physical device, an option explored in depth was the emulation of firmware.

**QEMU emulation**

Some of the previously mentioned challenges that occur when performing dynamic analysis can be overcome through the use of emulation. QEMU[[5]](#footnote-5) is a free and open-source emulator that utilises dynamic binary translation to emulate various processor architectures. It has multiple operating modes, but the ones that are applicable to IoT firmware emulation are user-mode emulation and system emulation.

In user-mode emulation, QEMU runs a single Linux binary that was compiled for a different instruction set than the host system. This has the advantage of being fast and easy to use, but it comes with the drawback of some binaries not working as intended if there is any peripherals required for the binary, and was found unsuitable for the purpose of general vulnerability discovery, although it still has some utility potential in testing some binaries and start-up scripts in the firmware. User mode emulation works through the use of a system call translator, which converts the parameters of the system calls to fix endianness issues and 32/64 bit mismatches between the host OS and targeted firmware.

System emulation emulates a full computer system including peripherals. Compared to user-mode emulation, system emulation creates a more accurate environment for penetration testing, as well as enables peripherals to be emulated, which is essential for a lot of binaries to run correctly. Emulating an internet adapter correctly can potentially enable a router webserver to be ran from the emulated environment, enabling it to be test through a fuzzer or other penetration testing software. A key disadvantage of system emulation is the fact that it is significantly more difficult to use and requires a matching Linux kernel and QEMU image containing the router filesystem. QEMU provides an image building utility that allows an image to be built relatively simply, but a Linux kernel is generally difficult to cross compile correctly due to the number of settings that need to be modified to make it compatible with both QEMU and the firmware being tested. Additionally, due to the time it takes to compile a kernel, it is not practical to compile one for every firmware analysed. Luckily Linux kernels are generally backwards compatible[[6]](#footnote-6), so a precompiled kernel for each architecture would be suitable for performing full system emulation on most firmware, as the kernels used when developing router firmware are generally pretty old. A major challenge with system emulation on router firmware is the fact that QEMUY doesn’t natively emulate nvram, an essential peripheral for a lot of binaries in the firmware. In a physical device, nvram is typically flash storage that holds the start-up configuration of the router and is typically interacted with nvram\_get() and nvram\_set() calls by the firmware binaries. If those nvram values are missing, the router in unable to initialise properly, and it is impossible to run the webserver, significantly reducing the usefulness of the emulation. Thus, a major challenge to overcome is the need for nvram if large scale dynamic analysis of firmware is to be achieved.

**Firmadyne**

Firmadyne[[7]](#footnote-7) is a tool that attempts to automate the process of embedded firmware analysis through the use of full system emulation. It handles the entire process from extracting firmware, creating a QEMU image and running an emulated device. Binwalk uses a custom extractor based on binwalk, as the basic recursive binwalk was not powerful enough and encountered issues when extracting some firmware. It then analyses the extracted binary to determine the architecture required to be emulated, builds a QEMU image from the extracted firmware and runs an initial emulation with a precompiled kernel. It then attempts to run the init binary in the firmware and analyses internet connections in the emulation to determine the required network peripherals. After the network is inferred, it runs the emulation again, this time with the correct network configuration, and at that point a webserver should be started. Unfortunately, during the testing performed, firmadyne seems to fail to infer network correctly the vast majority of the time. In fact, it was only able to successfully run one router firmware from the ~15 firmwares from different vendors that were tested. Most of the failures seem to have occurred during the infer network stage, with the error logs suggesting that come values in the nvram could not be accessed. Firmadyne handles nvram by compiling the nvram library, and then using LD\_PRELOAD to intercept calls to nvram\_set and nvram\_get in order to return the default values obtained from the firmware binary. The errors suggested that the nvram library did not have the required variables listed in it, which can be explained by the fact that firmadyne has not been actively updated since 2016 and most of the firmware tested was fairly recent, so it is likely that some additional configuration variables were added. Additionally, on the research paper by Daming D. Chen and his team[[8]](#footnote-8), from the 8000 embedded firmware tested, only around 2000 could be successfully fully emulated, as shown in figure 1.

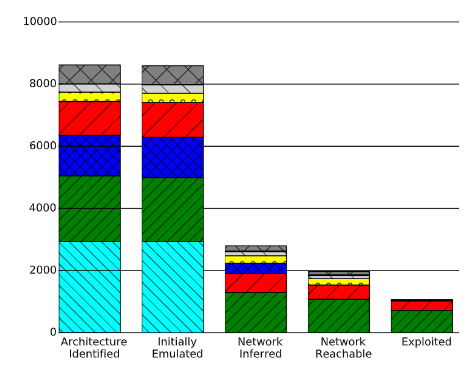


Figure Breakdown of firmware images by emulation progress, coloured by vendor (Daming D. Chen et al.)

Additionally, it is not known how much of the firmware tested were internet routers, which could potentially be more difficult to emulate than other embedded devices. The lack of coverage means that firmadyne needs to be manually updated to work with newer firmware if it is to be useful for analysis of router firmware, which is difficult to implement within the scope of this project. Another frequent error seems to be the QEMU image not being built correctly, and thus the initial emulation fails, so another method to circumvent this is required.

**Firmware Analysis Helper**

The project was initially focused on performing vulnerability analysis on the firmware of a specific router, the Archer A7. However after attempting to emulate it using QEMU in both user and system mode, as well as firmadyne, it became clear that to get it to emulate successfully, it was necessary to take a step back and refine the emulation environment by developing new tools to assist in the emulation.

The tool that was developed is based partly on a mix between the QEMU full system emulation method written in a blogpost by Vincent Lee[[9]](#footnote-9) and the implementation of firmadyne’s libnvram.

Firmware Analysis Helper is a dynamic analysis tool for Linux firmware images that was created as part of this project. Compared to firmadyne, the tool aims to achieve a higher success rate, as well as a faster execution time with less manual input, while specialising in specifically router firmware. A fundamental implementation difference is the fact that the tool does not attempt to build a QEMU images, and instead transfers the root file system of the firmware into an emulation based on a precompiled generic kernel and disk image of a similar architecture, and then using chroot to simulate the firmware environment.

The architecture of the tool is shown in figure 2.

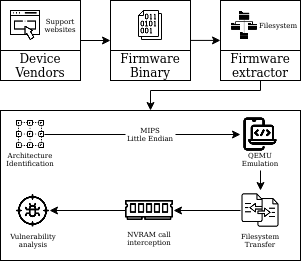


Figure FAH architecture

**Setup**

The tool must be installed using a setup script provided. The script downloads the precompiled kernel and images for every architecture, installs binwalk and other dependencies and creates the required directories. This was tested on both Ubuntu 18.04 and 22.04 and worked with no issues on either. Other version of Linux may require manual adjustment during installation or could be incompatible with some of the required packages. The precompiled packages are not included in the tool itself due to the relatively large file size of those systems. A potential future change could be to include compressed copies of the kernel and image to ensure that the tool is not dependent on third party sources for the kernel and QEMU images. Once setup is complete the tool can be ran using the start.sh script.

**Acquisition**

At the moment there is no automated method used to acquire router firmware. For the purposes of testing, firmware was downloaded from vendor websites, however a web scraper could also be used for large scale firmware analysis.

**Extraction**

The extraction uses recursive binwalk extraction, which proved to be powerful enough for all router firmware tested. In the research paper by the authors of firmadyne, it was mentioned that they found binwalk to occasionally fail to properly extract files or get stuck in an infinite extraction loop, however this was not encountered during this project, suggesting that it may only occur on specific types of embedded devices, as firmware analysis helper only aims to perform dynamic analysis on routers rather than any embedded Linux device.

**Architecture identification**

In order to perform system QEMU emulation, the architecture of the firmware needs to be identified. This is done by automatically analysing the extracted squashfs-root of the firmware, searching for the busybox binary, and then running the file command against it. The output is then analysed and searched for specific keywords like MIPS or MSB, which can be used to select the correct QEMU launch command and parameters.

**Emulation**

The QEMU system emulation is started by a shell script based on the determined architecture of the firmware. The precompiled images and kernels are obtained from a database by aurel32[[10]](#footnote-10), who created a publicly available database of QEMU-ready kernels and images for various architectures. Additional parameters are added t the launch command to enable a network adapted, with the device used being e1000. The i82557b network adapter is used instead during Armel emulation due to an incompatibility of the specific kernel or image used with the e1000 device when using an emulation of that architecture. Additionally, host forwarding rules are added to the emulation command to enable ssh functionality required for the tool and other general-purpose options like -nographic are used. Once the emulation is running, through the use of another script, the file system, the nvram library and another script are transferred into the emulation through ssh. The transferred script is then used to move the nvram library into the file system, and chroot is used to enter the root filesystem of the firmware. Additionally, LD\_PRELOAD is used to intercept calls to nvram with the transferred nvram library. Interestingly, little endian ARM cpu architecture was not able to successfully use the e1000 adapted. It is likely that this is a bug specific to the prebuilt QEMU image and kernel used, so it is difficult to fix without recompiling another kernel. To circumvent this issue, the intel i82557b network adapted is emulated instead. This adapter has a lower maximum internet speed than the e1000, however it is suitable for the purpose of hosting a webserver.

**NVRAM**

A modified version of firmadyne’s libnvram.so[[11]](#footnote-11) is used to intercept calls to nvram. This is currently in the process of being implemented, but it is planned to expand the range of nvram calls that it can intercept to include the parameters used by newer firmware. Firmadyne hasn’t been in active development since 2016, and the nvram library has received very little to no updates since then, resulting it in not working for the vast majority of modern router firmware.

The library works by reading the default config values in the firmware and compiling a library that uses these values. LD\_PREDLOAD is then used to intercept nvram calls and uses the library to return a reasonable value. This effectively emulates the physical component, whose main purpose is to store and retrieve the configuration of the router, and it is required for most tested binaries in router firmware.

**Vulnerability discovery**

The tool is designed to leverage other vulnerability discovery tools like fuzzers, Metasploit and Ghidra to perform dynamic vulnerability analysis. As of now, this hasn’t been tested due to nvram not being in a working state, however once the webserver of the router is operational, its pretty simple to perform analysis on it. Additionally, the tool can be used to test vulnerabilities in specific binaries discovered through other means such as static analysis to verify that those are actual vulnerabilities and haven’t been patched by an obscure or poorly decompiled piece of code.

**Discussion**

The implemented functionality of the tool works as expected, and the tool is able to start an emulation for MIPS and ARM architectures, both big and little endian. Some binaries that do not require nvram to work as intended are executable and can be tested, either manually or through a fuzzer, but the webserver has not successfully been booted for any of the firmware tested. The results of this project are difficult to quantify, as the tool is not complete and is not usable for most binaries at the moment, however even in its barebones state, the emulated environment is more powerful than user-mode QEMU emulation and is much easier to use at a large scale than manually determining the correct parameters required to run system emulation for each individual firmware.

Interestingly, the tool failed to start an emulation if the -nographic command was used, but only with a specific architecture and endianness, big endian ARM. It works fine otherwise, so it is unclear what causes this issue, as the -nographic option should not change the functionality of QEMU, only changing it so that it runs in terminal, rather than opening a new window. I estimate that it is a bug with QEMU when emulating that specific kernel version, but this is something that could be investigated later. There is very little difference in terms of emulation environment when using the -nographic option, so when emulating that processor architecture, FAH simply avoids using the option.

**Limitations**

The tool has a number of limitations that will be addressed in this section. A key limitation in using this tool is that even when it is setup to work correctly in the future, the nvram library must be manually updated to account for different firmware requiring different nvram calls. This means that it must be tested on a huge range of router firmware to improve coverage, as well as constantly updated to work with new firmware releases. A potential solution to this would be to create a framework that easily modifies the nvram library, reducing the amount of work required to correctly modify and recompile the library.

Another significant limitation is the inability to extract and emulate encrypted firmware. This limits the usefulness of the tool in a defence context, due to a presumably high level of encryption on custom embedded devices. There are methods to decrypt such firmware, but generally they require significant manual work. Encrypted routers often don’t start off having encrypted firmware. A lot of the time, encryption is implemented in a later firmware version, while one or more older versions contain information required to decrypt the newer versions. With this method, when the router updates to a newer, encrypted version of firmware, the old version has the tools required to decrypt it, so that it can be installed. One method to analyse such encrypted firmware is to obtain an older version that contains the method needed to decrypt the firmware, and then using it to decrypt the newer firmware. Generally, the encryption does not change between versions of encrypted firmware, so the original decryption binary or method can be used for any encrypted version[[12]](#footnote-12). This doesn’t work on routers that had their initial release encrypted, and instead use a decryption tool located in the hardware by default, so there are some limitations to this method.

Additionally, the emulated environment is not perfect. Since a precompiled kernel and image are used, the emulation is only an approximation of the real system. This is not an issue in the majority of cases, as Linux kernels are generally backwards compatible so simply using a newer kernel would avoid most problems, but this still could cause incorrect behaviour for come binaries.

Finally, it is likely that due to the lack of standardisation in embedded devices, some of the firmware used will fail to emulate correctly even with the nvram library set up correctly. This is a shortcoming of the QEMU framework itself and there isn’t a lot that can be done to circumvent this, so it’s expected that the coverage is not going to reach 100%.

**Future Work**

The first step for further development would be to get nvram to work correctly, as it is a requirement for the tool to function as intended. Once that is complete, and the tool is able to launch a webserver for most router firmwares, other features could be developed and improved upon, such as allowing GDB to analyse the emulation, enabling in depth debugging and reverse engineering. Additionally, this would help in fixing any bugs in the nvram library or the tool itself. Fully automating the tool is another potential angle for future work. At the moment the tool requires 3 scripts to be launched, and needs two terminals to be opened, one for the emulation, and one to transfer the filesystem to it. Ideally, the tool should only require a single command to run from start to fully emulated on any router firmware. Further work in large scale dynamic analysis would also likely require a method of automatically obtaining firmware images, so a web scraper that targets firmware binaries could be implemented to solve this challenge. Finally, actually using the tool to perform dynamic analysis and vulnerability discovery is something that could be done as future work. This can be achieved through the use of other vulnerability discovery methods such as fuzzing, debugging or using Metasploit.

References

[1] Techtarget [Online] Available: <https://www.techtarget.com/iotagenda/opinion/IoT-trends-to-keep-an-eye-on>

[2] Cloudflare [Online] Available: <https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/>

[3] Binwalk [Online] Available: <https://github.com/ReFirmLabs/binwalk>

[4] CVE list [Online] Available: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-42232>

[5] AFL++ [Online] Available: <https://aflplus.plus/>

[6] Metasploit [Online] Available: [www.metasploit.com](https://www.metasploit.com/)

[7] QEMU [Online] Available: <https://www.qemu.org/>

[8] Matt Rickard blogpost [Online] <https://matt-rickard.com/backward-compatibility>

[9] firmadyne [Online] <https://github.com/firmadyne/firmadyne>

[10] firmadyne research paper [Online] <https://github.com/firmadyne/firmadyne/blob/master/paper/paper.pdf>

[11] Vincent Lee “Mindshare: How to just emulate it with QEMU” [Online] <https://www.zerodayinitiative.com/blog/2020/5/27/mindshare-how-to-just-emulate-it-with-qemu>

[12] Midnshare: Dealing with encrypted router firmware [Online] <https://www.zerodayinitiative.com/blog/2020/2/6/mindshare-dealing-with-encrypted-router-firmware>

[13] Index of /~aurel32/qemu [Online] <https://people.debian.org/~aurel32/qemu/>

[14] firmadyne/libnvram [Online] <https://github.com/firmadyne/libnvram>

1. [GitHub - ReFirmLabs/binwalk: Firmware Analysis Tool](https://github.com/ReFirmLabs/binwalk) [↑](#footnote-ref-1)
2. [The AFL++ fuzzing framework | AFLplusplus](https://aflplus.plus/) [↑](#footnote-ref-2)
3. [www.metasploit.com](https://www.metasploit.com/) [↑](#footnote-ref-3)
4. [CVE - CVE-2021-42232 (mitre.org)](https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-42232) [↑](#footnote-ref-4)
5. [QEMU](https://www.qemu.org/) [↑](#footnote-ref-5)
6. [Backward Compatibility (matt-rickard.com)](https://matt-rickard.com/backward-compatibility) [↑](#footnote-ref-6)
7. [GitHub - firmadyne/firmadyne: Platform for emulation and dynamic analysis of Linux-based firmware](https://github.com/firmadyne/firmadyne) [↑](#footnote-ref-7)
8. [firmadyne/paper.pdf at master · firmadyne/firmadyne · GitHub](https://github.com/firmadyne/firmadyne/blob/master/paper/paper.pdf) [↑](#footnote-ref-8)
9. [Zero Day Initiative — MindShaRE: How to “Just Emulate It With QEMU”](https://www.zerodayinitiative.com/blog/2020/5/27/mindshare-how-to-just-emulate-it-with-qemu) [↑](#footnote-ref-9)
10. [Index of /~aurel32/qemu (debian.org)](https://people.debian.org/~aurel32/qemu/) [↑](#footnote-ref-10)
11. [GitHub - firmadyne/libnvram: NVRAM emulator](https://github.com/firmadyne/libnvram) [↑](#footnote-ref-11)
12. [Zero Day Initiative — MindShaRE: Dealing with encrypted router firmware](https://www.zerodayinitiative.com/blog/2020/2/6/mindshare-dealing-with-encrypted-router-firmware) [↑](#footnote-ref-12)