Wireless Networks
https://doi.org/10.1007/s11276-023-03517-z

Harden-IoT: hardening the EoL devices by intercepting the attack vector for future B5G/6G IoT

Xixing Li[1] · Qiang Wei[1] · Zehui Wu[1] · Wei Guo[1] · Linhao He[1]

Accepted: 8 September 2023

Abstract
Wireless networking provides many advantages, but it also coupled with new security threats and alters the organization's overall information security risk profile. Meanwhile, researchers are also actively working to enhance the security of embedded devices. However, previous studies have overlooked the insecurity of a particular device category, known as End-of-Life (EoL) devices. When a product reaches its EoL phase, vendors discontinue its maintenance, including the provision of bug fixes and security patches. However, many EoL devices remain on the internet with several critical vulnerabilities, thereby creating a fertile ground for botnets and cyber-attacks. Due to the lack of security updates, hardening the potentially vulnerable firmware in IoT devices is the most direct and promising defense method, but it has not been fully explored. In this paper, we propose a systematic active defense approach to harden EoL IoT devices, utilizing a hybrid binary rewriting method to monitor high-risk APIs and filter attack vectors. The proposed system, called Harden-IoT, consists of three tightly coupled modules: suspicious code snippets location, attack vector interception, and heuristic firmware repackaging. It can reinforce different architecture (MIPS, ARM) Linux-Based IoT devices without source code. We evaluate the effectiveness, adaptability, and overhead of the method using 23 firmware images sourced from various vendors. The results show that Harden-IoT can effectively defend against multiple types of vulnerabilities under low overhead conditions while also being able to adapt to different heterogeneous devices.

Keywords EoL devices · B5G/6G network · A systematic active defense · Cyber attacks

1 Introduction

Internet-Of-Things (IoT) devices play a vital and integral role in our daily life [1–3]. With emerging technologies such as artificial intelligence, THz and quantum communications, the development of "B5G/6G" has gained momentum in recent times [4]. It aims to promote IoT to Internet of Everything (IoE) and B5G/6G network is the footstone of smart cities. No doubt B5G/6G IoT will play an import role to support the applications of telemedicine, haptics, and connected autonomous vehicles, However, this raises concerns of escalating security risks in the tangible world [5]. A large number of IoT devices are vulnerable [6] and susceptible to attacks, as past incidents have shown [7]. Furthermore, due to various reasons (e.g., the development of new technologies or irresponsible manufacturers) [8], many IoT devices have become End of Life (EoL) devices [9], meaning no security updates in the future. Since a significant number of EoL devices remain on the Internet for an extended period along with numerous vulnerabilities, they have become a breeding ground for botnets and large DDoS attacks [10], seriously threatening cybersecurity [9].

Firmware Hardening is a promising method to improve the current security situation, but it has not been fully explored. The existing security work in the IoT field is classified into two categories—vulnerability discovery [11–13] and intrusion detection [14, 15]. However, both are unable to directly improve the current security situation [16, 17]. Although the efficiency of detecting vulnerabilities has been increased by the research on vulnerability mining,

✉ Qiang Wei
prof_weiqiang@163.com

Xixing Li
lxx.scholar@gmail.com

[1] National Digital Switching System Engineering and Technological Research Center, Henan 450000, Zhengzhou, China

Published online: 14 October 2023

hackers may also adopt these methods to mine more vulnerabilities for use in attacks. Even if well-intentioned researchers report the vulnerabilities to the manufacturers, it is still unknown whether they will responsibly patch them. Without security updates, reinforcing the potentially vulnerable firmware in IoT devices is the most direct and promising defense method. However, the current academic attention to this field of research is still very limited, far less than the previously mentioned vulnerability mining and intrusion detection.

Hardening the End of Life (EoL) IoT devices faces many challenges. First, the development environment in IoT devices is heterogeneous [18], hindering the implementation of generic defense. Second, various types of vulnerabilities are present in IoT devices, making the factors considered complex [19]. Third, IoT devices usually are equipped with constrained resources and lack secondary development interfaces [20], making it challenging to apply defensive solutions that have been proven effective on traditional computing platforms [21]. Subject to such challenges, existing approaches have drawbacks regarding defense capabilities and range scope. Kelly and Pitropakis [3] demonstrated that incorrect configuration is the core reason why IoT devices are susceptible to Mirai botnet attacks. RevARM [22] and ARMPatch [23] both attempted to use binary rewriting methods to patch IoT devices. Still, their consideration of security defense was not comprehensive enough, and their architecture was limited to ARM. Hadar and Siboni [24] propose an innovative cloud-based framework for protecting IoT devices, but this framework introduces additional devices, and its defense method heavily relies on the technical details of vulnerabilities, which is generally unavailable.

To address existing drawbacks, we design a systematic approach based on active cyber defense [23–25] to harden the EoL devices. Our Core idea is to apply the concept of active defense on general-purpose computing systems to low-end IoT devices, utilizing binary rewriting to monitor high-risk APIs and filter attack vectors. We developed a cross-platform approach, called Harden-IoT, that can reinforce different architecture (MIPS, ARM) Linux-Based IoT devices without source code. Harden-IoT consists of three tightly coupled modules: suspicious code snippets location, attack vector interception, and heuristic firmware repackaging.

We evaluated the effectiveness, adaptability, and overhead of our methodology using 23 firmware images sourced from various vendors such as D-Link, Netgear, Linksys, TP-LINK, and Cisco. Results show that Harden-IoT can effectively defend against multiple types of vulnerabilities under low (36.5%) overhead conditions and adapt to different heterogeneous devices.

The contributions of this paper are as follows:

1. We have designed a universal method that implements active defense on low-end EoL devices through binary rewriting.
2. Based on tests conducted on devices of different brands and models, we evaluated the defense capability of our method.
3. To the best of our knowledge, this is the first reinforcement method that ordinary consumers can practically use without expert knowledge, and without requiring the involvement of vendors.

2 Background

This section aims to demonstrate the importance of our research. Firstly, we elaborate on the weak security situation in the current IoT field. Then, some related work is listed and analyzed for its shortcomings.

2.1 Security status of IoT device

In recent years, the Internet of Things has flourished, bringing various conveniences to production and daily life through these devices. According to Statista [3], there were 23.14 billion connected IoT devices worldwide in 2018, which is projected to increase to 75.44 billion by 2025 [4]. Poor code robustness and lack of basic security measures [26, 27] make IoT devices the primary target of network attacks [28].

For example, in 2016, a large-scale cyber-attack occurred where attackers exploited vulnerabilities in IoT devices to connect them to a massive botnet, Mirai [29], which was then used to launch targeted attacks against networks. This attack caused a global network outage and severely impacted various sectors such as finance, manufacturing, healthcare, and public services.

The constantly-emerging vulnerabilities in IoT provide attackers with the possibility to control devices in bulk, while the security and privacy of users heavily rely on security patches from manufacturers [1]. However, the fact remains that manufacturers always lag behind attackers in fixing these vulnerabilities.

To make matters worse, due to reasons such as the release of new products, manufacturers going out of business, and irresponsibility, many IoT devices are abandoned by their manufacturers [9]. Manufacturers will no longer provide security patches for these IoT devices, and they are referred as End of Life (EoL) devices. As the lifecycle of electronic devices is long, research has shown that there are

still a significant number of EoL devices connected to the Internet, and the vulnerabilities on these devices not only provide breeding grounds for large-scale network attacks but also seriously endanger the security and privacy of countless consumers.

Furthermore, as time goes on, a significant number of devices will become vulnerable EoL devices. Therefore, it is a crucial research direction to enhance the security protection capabilities of EoL devices.

2.2 Related work

However, in the currently intensively studied field of network security, there is rarely work dedicated to hardening EoL devices to improve the current security situation. Prior works mainly focused on vulnerability discovery. The following are representative works in the field of vulnerability discovery.

In 2013, RPFuzzer [30] was a fuzzing framework specifically designed for finding protocol vulnerabilities for router devices.

In 2018, IoTFuzzer [31] leveraged the companion mobile apps of IoT devices to perform efficient black-box fuzzing.

In 2019, FIRM-AFL [32] was presented as a high-throughput grey box fuzzer for firmware running a POSIX-compatible operating system through augmented process emulation.

In 2019, DTaint [33] adopted pointer alias analysis to improve the data flow analysis and utilized data structure similarity matching to construct data dependence between functions invoked by indirect calls.

In 2020, KARONTE [34] was a static analysis framework for embedded firmware that can discover vulnerabilities due to multi-binary interactions. The authors achieve this goal by modeling and tracking multibinary interactions.

In 2021, SaTC [14] performed a taint analysis to discover bugs. It utilizes shared keywords related to user input in the front-end and back-end to infer the taint source.

Even though the academic community continues to make new advances in vulnerability discovery, these vulnerabilities may not be disclosed and fixed responsibly. Hence, the eradication of harm induced by vulnerabilities in cyberspace has emerged as a significant research question, with defense being commonly regarded as the most efficacious approach to tackle it.

Besides the work of vulnerability mining mentioned above, the following tasks aim to defend the attack from hackers.

In 2020, Kelly and Pitropakis [3] analyzed how Mirai infects IoT devices and proposed six defense measures from the perspective of service configuration. The experiment showed that devices are vulnerable to Mirai attacks with the manufacturer's default configuration.

In 2017, RevARM [22] was designed to accurately rewrite ARM binaries without source code across various platforms; it could be used for security applications.

In 2021, Huang and Song [23] designed a static binary patching solution suitable for vulnerability mitigation on ARM platforms.

However, these defense works are either insufficient in application scope or in defensive capacity. They fail to solve the reinforcement of a wide range of EoL devices to resist hacker attacks. So, this paper aims to fill this gap, designing a novel method to hardening the EoL devices.

3 Methodology

This section describes how our method bridges the aforementioned gaps. Firstly, we model the threat of EoL devices to determine the scope and problem boundaries of our research. Subsequently, we provide a theoretical overview of our approach.

3.1 Threat model

Based on previous research [1, 27] by numerous scholars, we have modeled the threats to IoT devices as follows.

1. Attackers can manipulate the packets of network requests to achieve remote interaction with IoT devices [35], but they cannot directly interact with the device through physical means [36].
2. Multiple types of vulnerabilities exist in IoT devices [37], including Buffer OverFlow (BoF), Command Injection (CI), and Information Leaking (IL), which are closely related to the use of sensitive functions.

Besides considering the above-mentioned threats, we also clarify the research subject. The most widely used Linux-Based devices are selected as our research targets. Our primary target is EoL devices, and thus we mainly focus on defending known vulnerabilities but also apply to the protection against 0 day vulnerabilities.

To broaden the applicability of our hardening solution, we have established two hypothetical conditions which the majority of devices are able to meet:

1. Regular users can utilize the firmware update feature embedded in their devices to upload new firmware for the purpose of updating [38].
2. Moreover, the firmware installed in EoL devices is predominantly unencrypted, thereby harboring the potential for repackaging [39].

3.2 Challenges

Upon determining the threat model, this section will scrutinize the challenges present in this field.

Challenge 1 The multitude of vulnerability types in IoT devices makes it an exceptionally challenging task to devise effective defenses against a variety of vulnerabilities.

Challenge 2 Designing a universal approach to accommodate a large number of fragmented Internet of Things (IoT) devices is a challenging task due to their wide heterogeneity and lack of secondary development interfaces.

Challenge 3 Effectively defending IoT devices under low-load conditions is a formidable challenge due to the constraints in resource availability.

3.3 Our solutions

After defining the threat model, this section expounds on our proposed solution. Based on the review, it is evident that EoL devices possess numerous types of vulnerabilities, many of which are closely associated with the use of sensitive Application Programming Interfaces (APIs) [9]. Therefore, our **core insight** is to monitor sensitive APIs and capture attack vectors before a vulnerability is exploited and subsequently block them to safeguard the devices. This idea requires the use of binary rewriting to modify the closed-source program. However, due to the challenges mentioned earlier, such as diverse IoT firmware and more complex development environments, existing binary rewriting methods may not sufficiently meet our requirements. Thus, we have developed a hybrid binary rewriting approach that combines high-level language with machine code, utilizing widespread dynamic loading mechanisms present in IoT devices to support our hardening solution better. After rewriting the original program, we employed a heuristic firmware repackaging method to package the program into new firmware, making it easy for end-users to utilize our solution.

4 System design

Based on the aforementioned solution, we designed a prototype system named Harden-IoT, As shown in the Fig. 1, it consists of three closely related modules: Suspicious Code Snippet Location, Attack Vector Interception, and Heuristic Firmware Repack.

The first module, Suspicious Code Snippet Location, is designed to identify vulnerable code snippets in the firmware. This module generates a list of all suspicious code locations.

The second module, Attack Vector Interception, takes the output from module ❶ as input and aims to detect and intercept attack vectors by monitoring the execution of suspicious code snippets. This module generates a hardened binary program.

The third module, Heuristic Firmware Repackaging, takes the output from module ❷ as input. As a response to the limitations of the current tool, it has been designed with 6 optimization measures. This module repackages the modified file system into a new firmware.

4.1 Suspicious code snippet location

In this module, the firmware image is received as the input. Instead of using traditional tools such as binwalk, we use the more powerful unpacking tool unblob [40] to extract the file system (usually named as rootfs) from the original firmware. unblob is an accurate, fast, and easy-to-use extraction suite which release on Github with 1.8k stars.

Because the most vulnerable targets are network service programs, we then traverse rootfs to locate their executable files after unpacking. Our protected targets include well-known services such as HTTPD and UPnP, as well as vendors' private service programs such as cfgserver in ASUS and HNAP in D-Link. Additionally, we consider Common Gateway Interface (CGI) [41] programs as protected targets due to the potential for multiple types of vulnerabilities and their invocation by web services. Following acquisition of the binary files of our protected targets, a static analysis is performed on them.

Static analysis, due to its inherent limitations, cannot simultaneously maintain low false negative and false positive rates. As our aim is defense, it is crucial to strike a balance between detecting as many possible vulnerabilities as possible and reducing ineffective monitoring. The following shows how we achieve this goal. Firstly, not all functions require monitoring. We first defined some sensitive APIs based on the descriptions in the literature [14, 42], as shown in Table 1.

As shown in Table 1, when sorting sensitive APIs, we considered not only common dangerous functions and also some vendor-defined functions that involve risky operations. Additionally, we associated sensitive APIs with their corresponding vulnerability types. We considered five types of vulnerabilities: Command Injection (CI), Information Leak (IL), SQL Injection (SQLI) and Buffer OverFlow (BOF). All calling points of these sensitive APIs will be located, but it is clearly impossible that all these calling points are vulnerable. So, to avoid unnecessary instrumentation and improve the efficiency of reinforcement, we need to recognize the high-risk sensitive APIs calling points through further analysis.

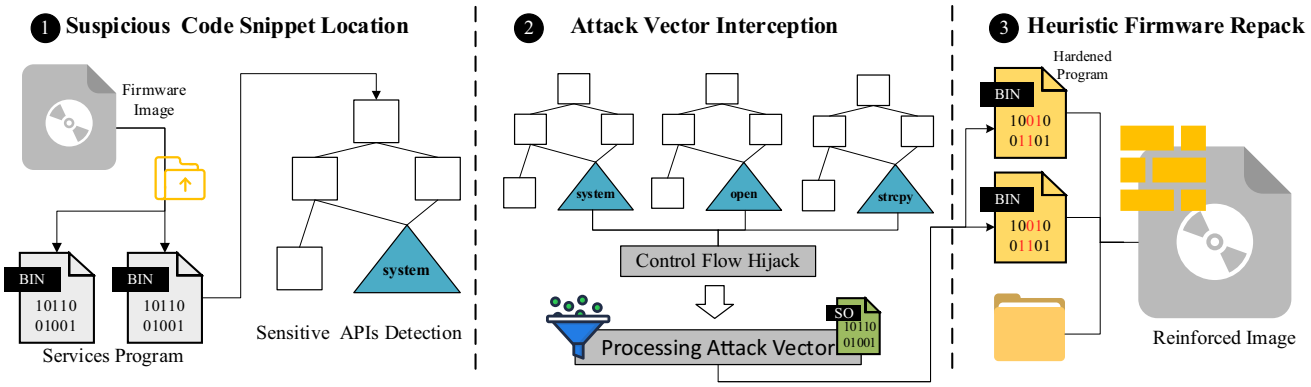


Fig. 1 Overview of Harden-IoT

Table 1 Info of Sensitive APIs

Seq	Vuln type	Sensitive APIs
1	Command Injection (CI)	system, execve, eval, dosystemcmd, lxmldbc_system, tp_systemEx, popen, COMMAND, SLIBCExec, SLIBCBackgroundExec
2	Information Leak (IL)	open, fopen
3	SQL Injection (SQLI)	sql_execute, SYNODBExecute
4	Format Strings (FST)	vsprintf, snprintf, fprintf
5	Buffer OverFlow (BOF)	strcpy, memcpy, sprintf, sscanf

$$S_{all} = \{S_i | S_i : Any\ Calling\ Points\ to\ Sensitive\ APIs\} \quad (1)$$

$$S_{safe} = \begin{cases} Fixed\ String\ s_i \in \{CI, IL, SQL, FST\} \\ Len(source) is\ fixed s_i \in \{BOF\} \\ Len(source) < Len(destination) s_i \in \{\text{BOF}\} \end{cases} \quad (2)$$

$$S_{risk} = S_{all} - S_{safe} \quad (3)$$

As shown in the Formula 1, S_{all} represents all calling points of sensitive APIs, and we defined the safe calling point S_{safe} if it satisfies any of the following three requirements:

(1) If the vulnerabilities types of sensitive APIs are **CI**, **IL**, **SQL**, **FST** and its arguments are fixed strings;
(2) If the vulnerabilities type of sensitive APIs is **BOF** and its length of source string is fixed;
(3) If the vulnerabilities type of sensitive APIs is **BOF** and its length is smaller than the destination buffer size.

After dropping out these "safe" calling points, the rest are recognized as high-risk calling points S_{risk}. However, the number of S_{risk} is still considerable, which will affect the efficiency of defense. Therefore, we furtherly designed a lightweight taint analysis algorithm to precisely locate the potential vulnerability point $S_{potential}$.

As the algorithm 1 shows, Initially, all external parameters are set as initial taint sources. Then, the decompiled code of each function in the program is traversed line by line. Each line of code is checked for direct assignment operations to variables. If the source variable is in the initial taint source, the target variable is added to the list of taint variables. When a risk point S_{risk} is encountered, the context of calling sensitive APIs is checked to determine whether there are any taint variables. If there are taint variables, the point is identified as a potential vulnerability point $S_{potential}$ and is added to the list and output.

Algorithm 1 : Suspicious Code Snippet Location

Input : list of S_{risk}.

Output : list of $S_{potential}$.

```
1 : function Taint_Analyse(risk_list):
2 :   for each_risk in risk_list:
3 :       all_potential ← ∅
4 :       addr = GetFuncAddr(each_risk)
5 :       all_potential+=Taint_Each_Func(addr)
6:    return all_potential
7 : function Taint_Each_Func(func_addr):
8 :   potential_list ← ∅
9 :   taint_vars_set = GetParams(func_addr)
10 :      func_code_content = GetDecompile(func_addr)
11 : for each_line in func_code_content:
12 :      if Is_Set_Variable(each_line):
13 :          src, dest=set_variable_info(each_line)
14 :          if src in taint_vars_set:
15 :              taint_vars_set.append(dest)
16 :          if Is_Risk_Point(each_line):
17 :              src_argv= get_sensitive_argv(each_line)
18 :              if intersection(src_argv, taint_vars_set)! = ∅:
19 :                  potential_list.append(each_line)
20 : return potential_list
```

4.2 Attack vector interception

4.2.1 Detect and intercept the attack vector

Once the potential vulnerability points $S_{potential}$ have been identified, we apply active defense techniques at these points to intercept the attack vectors. Firstly, we categorize the attack vectors, as shown in Table 2, as different attack vectors possess distinct characteristics. We employ different identification strategies to detect these attack vectors, and upon detection, intercept the trigger of vulnerabilities by removing the harmful parts within their parameters.

As shown in Table 2, attack vectors are classified into two types: Special Char and Strings Length. Special Char refers to the injection of special characters by attackers to break the security assumption and execute malicious code while exploiting vulnerabilities. The vulnerabilities involved include command injection, information leakage, SQL injection, and format string vulnerabilities. Strings Length refers to the use of very long strings by attackers to destroy memory boundaries and execute malicious code. The main vulnerability involved is buffer overflow.

For Special Char attack vectors, our detection method is string checking. During runtime, we monitor the execution of sensitive APIs, obtain their parameters, and then check for the presence of special characters in the parameters. If any special characters are found, an alert is issued, and those characters are removed to block the attack vector and protect the device from being compromised.

For Strings Length attack vectors, our detection method is buffer checking. Similar to the previous method, we obtain the parameters of sensitive APIs at runtime and check their length. If the length of the parameter exceeds the buffer length, an alert is issued, and the excessive part of the parameter is removed to block the attack vector and protect the device from being compromised.

4.2.2 Hybrid binary instrumentation

To deploy the aforementioned defense measures at these potential vulnerability points $S_{potential}$ without obtaining the source code of the program, we must rely on static binary instrumentation to inject the defense code into the original program. However, as we cannot obtain the source code of these programs, we must rely on static binary instrumen-

Table 2 Info of Attack Vector and Detection Strategy

Seq	Vuln type	Vector type	Judgement rules	Marked string
1	Command injection	Sensitive string	Substring check	;whoami;ps -a
2	Information leak	Special char	Substring check	../../etc./shadow
3	SQL injection	Special char	Substring check	(“,&,|,')
4	Format strings	Special char	Substring check	%n, %p
5	Buffer overflow	Strings Length	Buffersize check	check buffer size

tation to inject defense code into the original program. Current static binary instrumentation methods only support the use of restricted assembly programming by analysts to inject additional code. This requires a significant amount of expert knowledge and manual effort, making it difficult to scale to the large quantity of diverse IoT devices.

To address this issue, we have designed a novel instrumentation framework that allows the use of high-level languages to write injected code. This framework utilizes the widespread use of the dynamic loading mechanism [43] in IoT devices to combine high-level languages with machine instructions, providing high programmability and adaptability to support the implementation of our defense methods.

We believe that instrumentation can be divided into two core modules, execution flow hijacking and injected code generation. Existing static analysis methods bind these two core modules together and implement them using pure assembly programming, which is the main reason for their poor adaptability and low programmability.

Therefore, our new framework separates these two core modules and continues to use assembly language to support execution flow hijacking. We then innovatively use high-level languages to write injected code, which is compiled into a shared library using cross-compilation tools. This shared library is loaded using the dynamic loading [43] mechanism widely available in IoT devices, thus achieving an organic integration between high-level languages and assembly language.

As the figure shows, the whole instrumentation process can be divided into three steps: expansion, hybrid instrumentation and establishment of shared library.

Expansion it aims to enlarge the space of original binary for following inserted machine instructions.

First, it creates a new section or modify an unused section within the ELF structure of a program binary to add a new memory space that is accessible for reading, writing, and executing (RWX). Then, a new entry is added in the**.dynamic** section for loading the extra self-made shared library. After that, the extended binary has a large enough memory space to afford the extra instructions for instrumentation. In addition, it could load a self-made dynamic module which could be written in high-level language at runtime.

Hybrid Instrumentation To link high-level languages and reduce repetitive instructions, as shown in Fig. 2, we optimize existing trampoline method, making the inserted instructions consist of three parts for accomplishing three tasks.

(1) the replaced instruction which hijack control flow at basic block;
(2) the direct transfer instructions which store run-time state before change in the following operation;
(3) the common transfer instructions which set the parameters for invoking defense primitives in the shared library written in a high-level language.

Through cross-compile, the high-level language source code turns into a dynamically loadable shared library and is loaded to memory through adding a new entry pointed to shared library in the. dynamic section of Executable and linkable Format (ELF).

Due to the aforementioned hybrid instrumentation method that requires compiling defense code, written in high-level languages, into shared libraries compatible with the original program, a suitable cross-compilation toolchain is necessary. However, in most cases, vendors do not disclose their original cross-compilation toolchains, making it challenging to obtain one. Therefore, we designed a method to automatically generate a suitable cross-compilation toolchain. First, we inferred the cross-compilation toolchain's configuration information from the firmware information, and then handed over these configurations to a universal cross-compilation tool generator to create a toolchain that matches the corresponding firmware.

This involves defining configuration information. Since the high-level language we use to write defense code is C, as long as the shared library can correctly call the C language API in the firmware, it will be compatible with the original program. Therefore, we define the configuration information for the cross-compilation tool as follows formula. Arch refers to the architecture of the target image, it could be ARM and MIPS; Endian indicates the order of bytes in memory, it could be little endianness or big endianness; Libc stands for the type and version of the standard C library used in the image. Float stands for the strategy used by the target device when handling floating point arithmetic.

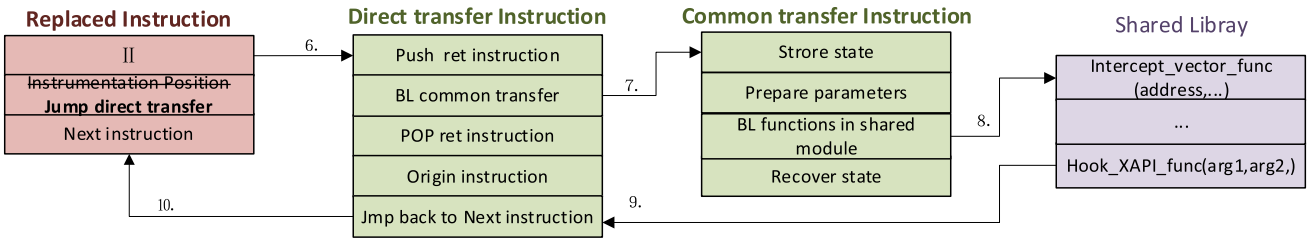


Fig. 2 Detail of Hybrid Instrumentation

$$Conf = <Arch, Endian, Libc, Float> \quad (4)$$

In the unpacking process, we use static analysis to infer this information, and then pass the information to the buildroot [44] tool to obtain a toolchain that is compatible with the original firmware.

4.3 Heuristic firmware repacking

IoT repackaging devotes to tampering with a legitimate firmware package by modifying its content and re-distributing it in the wild [39]. This technology is often abused by hackers to carry out various attack behaviors, but we will utilize this technology to persistently deploy our defense measures.

The state-of-the-art tool for firmware repackaging is Firmware-Mod-Kit (FMK), which often encounters many failures. Through extensive experimentation, we have identified the root causes of these failures. Drawing inspiration from these causes, we designed a novel heuristic repackaging method that incorporates five optimization measures to improve the success rate of automated repackaging.

1. **Perceiving and Maintaining the Original Structure of Firmware:** The rootfs component in firmware is the core of operations and our modifications are limited to it. However, rootfs is not the entirety of the firmware and is often located in the middle of the firmware. Existing tools ignore the original structure of firmware and the repackaged firmware may not pass the vendor's format verification. Therefore, when unpacking the firmware, we perceive its original structure and collect the header and footer of the original firmware for restoration during repackaging.
2. **Maintaining Consistency in Filesystem Packaging Details:** Rootfs has different filesystem packaging formats. When unpacking, we record the core parameters, such as block size and specific version number of the filesystem. During repackaging, we use the original parameters for rewriting and restoration to ensure consistency to the greatest extent possible.
3. **Maintaining Consistent Filesystem Properties:** Different directories within Rootfs have varying permission assignments and various symbolic link files. Existing tools may easily disrupt these structures. Thus, during repackaging, we check whether these properties are consistent with their original values.
4. **Minimizing Changes in Filesystem Size:** In scenarios of IoT devices with limited resources, the smaller the modifications made to the firmware, the higher the success rate of repackaging. Therefore, during repackaging, we focus on reducing the size of the additional code inserted. This is mainly achieved by using highly optimized compilation options during cross-compilation to reduce the size of binary files.
5. **Reducing Redundancy in Files:** In order to offset the increase in rootfs caused by security measures, we have compressed some redundant files, such as JS and PNG files used in WEB access. These files do not change their semantic meaning after compression, and thus do not affect the original functionality. By doing so, we ensure that the modified rootfs size is as close to the original size as possible, thus improving the success rate of repackaging.

5 Evaluation

We evaluate Harden-IoT with three tasks to answer the following research questions:

RQ1 **Effectiveness**, Can Harden-IoT defense various type vulnerabilities?

RQ2 **Adaptability**, Can Harden-IoT adapted to heterogeneous IoT devices?

RQ3 **Low Overhead**, Does Harden-IoT can defense IoT devices with low runtime overhead?

5.1 Effectiveness

We selected 13 vulnerabilities, including CI, BOF, SQLI, FST, and IL 5 types, associated with 11 devices from eight different vendors. For these 11 devices with vulnerable firmware, we conducted emulations and used Harden-IoT to harden the firmware. We then launched attacks on both the original and reinforced firmware using exploit scripts

Table 3 Info of Evaluation Vulnerabilities

Vuln-ID	Type	Brand	Device	API	Exploit Result	
					Origin	Harden-IoT
CVE-2018–16333	BOF	Tenda	AC15	dosystemcmd	x	√
CVE-2020–10987	CI	Tenda	AC15	dosystemcmd	x	√
CVE-2020–15916	CI	Tenda	AC15	dosystemcmd	x	√
CVE-2020–8515	CI	Draytek	Vigor2960	popen	x	√
CVE-2020–14472	CI	Draytek	Vigor2960	popen	x	√
CNVD-2022–31604	IL	ASUS	RT56u	open	x	√
CVE-2021–1610	CI	Cisco	RV160	system	x	√
CVE-2020–3331	BOF	Cisco	RV110	sscanf	x	√
CVE-2019–16057	CI	D-Link	DNS320	system	x	√
CVE-2015–2051	CI	D-Link	DIR645	lxmldbc_system	x	√
CVE-2022–44251	CI	TOTOLink	NR1800X	dosystem	x	√
CVE-2019–20760	CI	NETGEAR	R9000	system	x	√
CVE-2021–33180	SQLI	Synology	DS220j	SYNODBExecute	x	√
CVE-2018–14713	FST	ASUS	AC3200	sprintf	x	√

Table 4 Info of evaluation devices for adaptability

Seq	Brand	Device	Type	LIBC	Float	Endian	Program
1	Cisco	RV110	MIPS	uclibc_0.9.33	Hard	Little	httpd
2	Tenda	AC15	ARM	uclibc_0.9.30	Soft	Little	httpd
3	D-Link	DIR-810L	MIPS	uclibc_0.9.28	Hard	Little	cgibin
4	D-Link	DIR-850L	MIPS	uclibc_0.9.28	Hard	Big	cgibin
5	ASUS	EA4500	MIPS	uclibc_0.9.33.2	Hard	Big	httpd
6	TP-Link	WR940N	MIPS	uclibc_0.9.30	Hard	Big	httpd
7	ASUS	AC56U	ARM	musl_1.2.3	Soft	Little	httpd
8	NETGEAR	R8000	ARM	uclibc_0.9.33	Soft	Little	httpd
9	D-Link	DIR-665	ARM	glib_2.5	Hard	Little	cgibin
10	Cisco	RV110	MIPS	uclibc_0.9.33	Hard	Little	httpd

capable of triggering each specific vulnerability. If a device was breached, the corresponding cell in the exploit result column was marked as "X", while if the attack was unsuccessful, it was marked as a "√". The experimental results are recorded in Table 3 which clearly shows that Harden-IoT has successfully defended against all vulnerabilities. This confirms the effectiveness of our defense measures.

5.2 Adaptability

To test the adaptability of Harden-IoT, this chapter selects 10 representative devices as experimental targets. As shown in Table 4, these devices include three architectures, ARM and MIPS, and multiple libc versions. We use Harden-IoT to reinforce these devices and observe whether they can operate normally after reinforcement. The experimental results show that Harden-IoT can be reinforced on all 10 devices, which demonstrates its sufficient adaptability.

5.3 Low-overhead

To measure the performance of Harden-IoT, we conducted the following experiments on 10 devices. The information of these devices is listed in the Table 4. We prepared 240 network requests, sent them to both the original and reinforced devices, and recorded their average response time. Changes in response time can be used to calculate the impact of reinforcement on device load. The experimental results are recorded in Fig. 3, where the rectangle pink color column represents the number of defense points arranged in the program, the rectangle light blue column represents the average response time of the original device, and the rectangle violet column represents the average response time of the reinforced device. And the device id in Fig. 3 is corresponding to the seq in Table 4. The experiment showed that the more defense points arranged in the program, the greater the increase in response time. However, the overall average data shows that the increase in average response

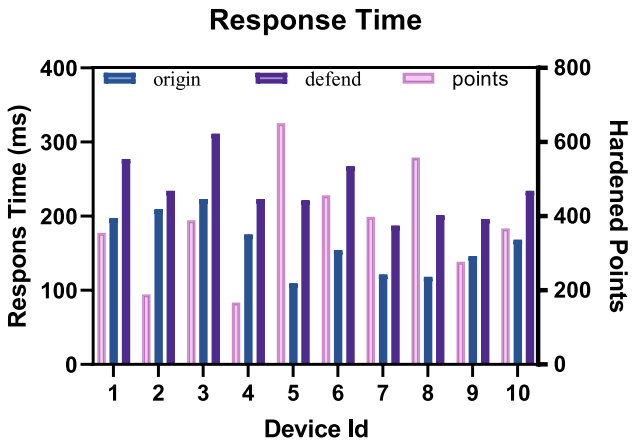


Fig. 3 Result of average response time on original devices and hardened devices

time is 36.5% less than 40%, which achieves low overhead on resource-limited IoT devices.

6 Discussion

In this section, we make a discussion about the limitation and capability of our method. Our Harden-IoT relies on the hybrid binary instrumentation method which is based on the dynamic loading mechanism. In theory, this method is applicable to all devices that support the dynamic loading mechanism. Among these devices, Linux-based devices are a significant category, and they have been selected as the subjects of our experiments. We plan to expand the scope of our system, Harden-IoT, in the future by integrating it with other systems like the Cisco IOS system, which also utilizes dynamic loading mechanism. Currently, our current methods can only defend against vulnerabilities found in binary programs. However, in order to enhance our defense capabilities, we intend to conduct research on protective methods specific to script programs in the future. By the way, hybrid binary instrumentation is a highly extensible method that allows users to customize the modified code. In the future, we will try to use this method to enhance fuzzing and reverse engineering.

7 Conclusion

There are millions of vulnerable IoT devices on the internet, and more and more of these devices are abandoned and become EoL devices. Previous security research has neglected the security risks associated with EoL devices and the necessity of hardening them. This paper summarizes the challenges of hardening EoL devices and innovatively applies the concept of active defense to resource-limited IoT devices, designing a defense method to intercept attack vectors. We implemented our prototype system, Harden-IoT, by using hybrid binary instrumentation and heuristic firmware repackaging. We conducted experiments to demonstrate the ability of Harden-IoT to protect effectively against various vulnerabilities when operating under low load circumstances. Additionally, it can adapt to a range of heterogeneous devices.

Data availability No dataset was generated or analyzed during this study.

Declarations

Conflict of interest All authors disclosed no relevant relationships.

References

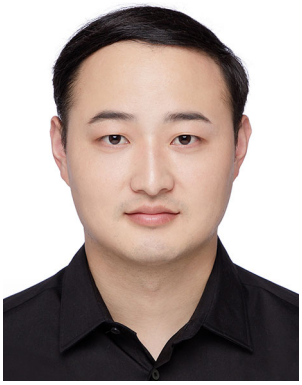
1. Nadir, I., Mahmood, H., & Asadullah, G. (2022). A taxonomy of IoT firmware security and principal firmware analysis techniques. *International Journal of Critical Infrastructure Protection*. https://doi.org/10.1016/j.ijcip.2022.100552
2. Trendmicro, "Mirai Botnet Attack IoT Devices via CVE-2020–5902," 2020. Accessed: Apr. 28, 2022. [Online]. Available: https://www.trendmicro.com/en_us/research/20/g/mirai-botnet-attack-iot-devices-via-cve-2020-5902.html
3. C. Kelly, N. Pitropakis, S. McKeown, and C. Lambrinoudakis (2020) "Testing And Hardening IoT Devices Against the Mirai Botnet," in *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, pp. 1–8. https://doi.org/10.1109/CyberSecurity49315.2020.9138887.
4. Qadir, Z., Le, K. N., Saeed, N., & Munawar, H. S. (2023). Towards 6G internet of things: Recent advances, use cases, and open challenges. *ICT Express, 9*(3), 296–312. https://doi.org/10.1016/j.icte.2022.06.006
5. Nguyen, V.-L., Lin, P.-C., Cheng, B.-C., Hwang, R.-H., & Lin, Y.-D. (2021). Security and privacy for 6G: A survey on prospective technologies and challenges. *IEEE Commun Surv Tutor., 23*(4), 2384–2428. https://doi.org/10.1109/COMST.2021.3108618
6. fraunhofer, "Home Router Security Report 2020," 2020. Accessed: Apr. 28, 2022. [Online]. Available: https://www.fkie.fraunhofer.de/en/press-releases/Home-Router.html
7. Microsoft, "Microsoft Digital Defense Report 2022," 2023. [Online]. Available: https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RE5bUvv?culture=en-us&country=us
8. Sivakumaran, P., & Blasco, J. (2021). argXtract: Deriving IoT security configurations via automated static analysis of stripped ARM cortex-M binaries. *ACSAC*. https://doi.org/10.1145/3485832.3488007
9. D. Wang *et al.*, "A measurement study on the (in)security of end-of-life (EoL) embedded devices," *CoRR*, vol. abs/2105.14298, 2021, [Online]. Available: https://arxiv.org/abs/2105.14298
10. SECTRIO, "The 2022 IoT and OT Global Threat Landscape Assessment Report," Feb. 21, 2022. https://sectrio.com/iot-security-reports/2022-iot-and-ot-threat-landscape-assessment-report/ (accessed Jun. 14, 2022).
11. Redini N., *et al.* (2021). "Diane: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San*

Francisco, CA, USA, 24-27 May 2021, IEEE, pp. 484–500. doi: https://doi.org/10.1109/SP40001.2021.00066
12. Chen, D. D., Woo, M., Brumley, D., & Egele, M. (2016). "Towards automated dynamic analysis for linux-based embedded firmware," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21–24, 2016*, in NDSS'16. The Internet Society. https://doi.org/10.14722/ndss.2016.23415.
13. Yun, J., Rustamov, F., Kim, J., & Shin, Y. (2022). Fuzzing of embedded systems: A survey. *ACM Computing Surveys, 55*(7), 1–33. https://doi.org/10.1145/3538644
14. Chen L., et al. (2021). "Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems," presented at the 30th USENIX Security Symposium (USENIX Security 21), in Security 21. pp. 303–319. Accessed: Apr. 24, 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-libo
15. Xie W., et al. (2022). "Game of Hide-and-Seek: Exposing Hidden Interfaces in Embedded Web Applications of IoT Devices," in *Proceedings of the ACM Web Conference 2022*, in WWW '22. New York, NY, USA: Association for Computing Machinery, pp. 524–532. doi: https://doi.org/10.1145/3485447.3512213.
16. Bagaa, M., Taleb, T., Bernabé, J. B., & Skarmeta, A. F. (2020). A machine learning security framework for iot systems. *IEEE Access, 8*, 114066–114077. https://doi.org/10.1109/ACCESS.2020.2996214
17. Thakkar, A., & Lohiya, R. (2021). A review on machine learning and deep learning perspectives of IDS for IoT: Recent updates, security issues, and challenges. *Arch. Comput. Methods Eng., 28*(4), 3211–3243. https://doi.org/10.1007/s11831-020-09496-0
18. Muench, M., Stijohann, J., Kargl, F., Francillon, A., & Balzarotti, D. (2018). "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Proceedings 2018 Network and Distributed System Security Symposium*, in NDSS'18. https://doi.org/10.14722/ndss.2018.23166.
19. Song D., et al. (2019). "SoK: Sanitizing for security," in *2019 IEEE symposium on security and privacy, SP 2019, san francisco, CA, USA, may 19-23, 2019*, IEEE, pp. 1275–1295. https://doi.org/10.1109/SP.2019.00010
20. Hawkins, W. H., Hiser, J. D., Co, M., Nguyen-Tuong, A., & Davidson, J. W. (2017) "Zipr: Efficient static binary rewriting for security," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, pp. 559–566. https://doi.org/10.1109/DSN.2017.27.
21. Mtetwa, N. S., Tarwireyi, P., Abu-Mahfouz, A. M., & Adigun, M. O. (2019). "Secure firmware updates in the internet of things: A survey," in *2019 International Multidisciplinary Information Technology and Engineering Conference (IMITEC)*, Nov. pp. 1–7. https://doi.org/10.1109/IMITEC45504.2019.9015845.
22. Kim et al. (2017). "RevARM: A platform-agnostic ARM binary rewriter for security applications," in *Proceedings of the 33rd annual computer security applications conference, orlando, FL, USA, december 4–8, 2017*, ACM, pp. 412–424. https://doi.org/10.1145/3134600.3134627.
23. Huang M., & Song, C. (2021). "ARMPatch: A binary patching framework for ARM-based IoT devices," *Journal of Web Engineering* pp. 1829–1852
24. Hadar, N., Siboni, S., & Elovici, Y. (2017). "A lightweight vulnerability mitigation framework for IoT devices," in *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*, in IoTS&P '17. New York, NY, USA: Association for Computing Machinery, pp. 71–75. doi: https://doi.org/10.1145/3139937.3139944.
25. Guo, B., Dian, S., & Zhao, T. (2022). Active event-driven reliable defense control for interconnected nonlinear systems under actuator faults and denial-of-service attacks. *Science China Information Science, 65*(6), 1–17. https://doi.org/10.1007/s11432-021-3397-2
26. Mera, A., Feng, B., Lu, L., & Kirda, E. (2021). "DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, IEEE, pp. 1938–1954. https://doi.org/10.1109/SP40001.2021.00018
27. Li, D., Hu, Y., Xiao, G., Duan, M., & Li, K. (2023). An active defense model based on situational awareness and firewalls. *Concurrency Computation Practice Experience, 35*(6), 1. https://doi.org/10.1002/cpe.7577
28. Palo Alto Networks., "2020 Unit 42 IoT Threat Report," Mar. 2020. Accessed: Apr. 28, 2022. [Online]. Available: https://unit42.paloaltonetworks.com/iot-threat-report-2020/
29. Yu, M., Zhuge, J., Cao, M., Shi, Z., & Jiang, L. (2020). A survey of security vulnerability analysis, discovery, detection, and mitigation on IoT devices. *Future Internet*. https://doi.org/10.3390/fi12020027
30. Wang, Z., Zhang, Y., & Liu, Q. (2013). RPFuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing. *KSII Transactions on Internet and Information Systems, 7*(8), 1989–2009. https://doi.org/10.3837/tiis.2013.08.014
31. Chen J., et al. (2018). "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing," in *Proceedings 2018 Network and Distributed System Security Symposium*, doi: https://doi.org/10.14722/ndss.2018.23159.
32. Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., & Sun, L. (2019). "FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation," in *28th USENIX Security Symposium*, pp. 1099–1114.
33. Cheng K., et al. (2018). "DTaint: Detecting the Taint-Style vulnerability in embedded device firmware," in *Proceedings - 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018*, in DSN'18. IEEE, pp. 430–441. https://doi.org/10.1109/DSN.2018.00052.
34. Redini N., et al. (2020). "Karonte: Detecting Insecure Multi-binary Interactions in Embedded Firmware," in *In Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, pp. 1544–1561. doi: https://doi.org/10.1109/sp40000.2020.00036.
35. Davidson, D., Moench, B., Ristenpart, T., & Jha, S. (2013). "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Proceedings of the 22th USENIX security symposium, washington, DC, USA, august 14–16, 2013*, S. T. King, Ed., USENIX Association, pp. 463–478. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson
36. Zhang, C., Wang, Y., & Wang, L., "Firmware fuzzing: The state of the art," in *12th Asia-Pacific Symposium on Internetware*, in Internetware'20. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 110–115. https://doi.org/10.1145/3457913.3457934.
37. Kim, M., Kim, D., Kim, E., Kim, S., Jang, Y., & Kim, Y. (2020). "FirmAE: towards large-scale emulation of IoT firmware for dynamic analysis," in *Annual Computer Security Applications Conference*, in ACSAC '20. New York, NY, USA: Association for Computing Machinery, pp. 733–745. https://doi.org/10.1145/3427228.3427294.
38. Feng, X., Zhu, X., Han, Q.-L., Zhou, W., Wen, S., & Xiang, Y. (2023). Detecting vulnerability on IoT device firmware: A survey. *IEEECAA Journal of Automatica Sinica, 10*(1), 25–41. https://doi.org/10.1109/JAS.2022.105860
39. Verderame, L., Ruggia, A., & Merlo, A. (2023) "PARIOT: Anti-repackaging for iot firmware integrity." arXiv, Jan. 25. https://doi.org/10.48550/arXiv.2109.04337.
40. O. developers, "unblob - extract everything!" https://unblob.org/ (accessed Jul. 10, 2023).

41. Gundavaram, S. (1996). *CGI programming on the world wide web*. O'Reilly & Associates
42. Cheng K., et al. (2022). "Finding taint-style vulnerabilities in linux-based embedded firmware with SSE-based alias analysis," *ArXiv*
43. O'Neill, R. (2016). *Learning linux binary analysis*. Packt Publishing.
44. Buildroot, "Buildroot - Making Embedded Linux Easy." https://buildroot.org/ (accessed Apr. 24, 2023).

Xixing Li received the Master's degree in computer science and technology from China National Digital Switching System Engineering and Technological Research Center, Zhengzhou, China, in 2019. He is currently a Ph.D. candidate with the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou. His current research interests include IoT security, Program analysis, and vulnerability discovery.

Qiang Wei received the Ph.D. degree in computer science and technology from China National Digital Switching System Engineering and Technological Research Center, Zhengzhou, China, in 2007. He is currently a Professor with the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou. His current research interests include network security, industrial Internet security, and vulnerability discovery.



Zehui Wu received the Ph.D. degree in software engineering from the China National Digital Switching System Engineering and Technological Research Center, Zhengzhou, China, where he is currently a Lecturer. His research interests include program analysis, reverse engineering, and SDN security.

Wei Guo received the Master's degree in computer science and technology from China National Digital Switching System Engineering and Technological Research Center, Zhengzhou, China, in 2022. He is currently a Ph.D. candidate with the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou. His current research interests include network security, Program analysis, and vulnerability discovery.

Linhao He received the master's degree in computer science and technology from China National Digital Switching System Engineering and Technological Research Center, Zhengzhou, China, in 2023. His current research interests include network security and vulnerability discovery.