DFirmSan: A Lightweight Dynamic Memory Sanitizer for Linux-based Firmware

Shanquan Yang[a,b], Yansong Gao[c], Boyu Kuang[d,b,*], Yixuan Yang[d], Anmin Fu[a,b]

[a]*School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, PR China*
[b]*State Key Laboratory of Integrated Services Networks, Xidian University, Xi'an, PR China*
[c]*Department of Computer Science and Software Engineering, University of Western Australia, Perth, Australia*
[d]*School of Cyber Science and Engineering, Nanjing University of Science and Technology, Nanjing, PR China*

Abstract

Vulnerabilities in Linux-based firmware present a significant risk to IoT security, with memory-related issues being especially hidden and dangerous. Despite substantial efforts to uncover firmware vulnerabilities through fuzzing, these methods are often ineffective in detecting memory vulnerabilities. To address this issue, prior research introduces sanitizers integrated into fuzzers. However, applying existing sanitizers to Linux-based firmware poses three significant challenges: First, embedded Linux systems lack robust memory protection and operate under tight performance constraints, making it difficult to detect "silent memory corruption." Second, most binary sanitizers focus on executables, such as the main program (the core backend service programs handling requests), and fail to effectively monitor dynamically loaded libraries, which are often assumed to be trustworthy. Third, sanitizers that rely on global memory monitoring techniques, such as shadow memory or redzone, introduce substantial performance overhead. These mechanisms significantly slow down resource-constrained firmware, rendering fuzz testing impractical for IoT devices. This paper introduces DFirmSan, a lightweight dynamic memory sanitizer for Linux-based firmware. DFirmSan addresses key challenges in detecting memory vulnerabilities through a two-step process. First, the pre-analysis phase identifies service programs and vendor-customized libraries, analyzing them for sensitive function calls and key parameters. In the second step, dynamic memory corruption detection, DFirmSan leverages this information to perform targeted dynamic boundary checks during runtime, focusing on detecting memory flaws, particularly silent corruptions. To minimize overhead, DFirmSan focuses on selectively monitoring sensitive function parameters influenced by untrusted data, rather than tracking all memory variables. It further reduces false positives by

*Corresponding author. Email: kuang@njust.edu.cn

dynamically adjusting parameter boundaries. We evaluate DFirmSan on 18 real-world firmware samples. By integrating DFirmSan, two advanced fuzzers detect 117 and 25 additional known CVEs, respectively. Besides, it helps uncover 4 CNVD zero-day vulnerabilities. Despite this enhanced capability, the impact on fuzzing speed remains minimal, with reductions of only 16.43% and 2.69%, well within acceptable limits. Moreover, DFirmSan maintains an impressively low false positive rate of under 0.35% for detecting memory corruption, further underscoring its practicality in real-world firmware.

Keywords:
Sanitizer, Linux-based Firmware, Internet of Things, Fuzzing

---

1. Introduction

Programs in real-world applications suffer from a variety of flaws, with memory corruption bugs being particularly common. To tackle this issue, numerous advanced fuzzers have been developed to identify memory vulnerabilities [1, 2, 3]. These fuzzers generate a large number of test cases for the target program and detect memory violation signals (e.g., crashes). However, solely reliance on the existing crash signals is insufficient to indicate all program flaws. For example, a buffer overflow vulnerability may not trigger a crash in many cases [4]. Thus, to design a solid fuzzer, a major challenge is how to capture these security violation signals efficiently, accurately, and holistically.

In this context, many memory sanitizers are designed to enhance the ability of fuzzers to detect program flaws on a variety of platforms, including desktop system [4, 5, 6, 7] and bare-metal [8, 9]. Based on the working mechanisms of existing sanitizers, they can be classified into two categories: static sanitizers and dynamic sanitizers [10]. Static sanitizers [4, 6, 7, 8] instrument source-code or compiled binary, to embed specific detection code for runtime memory checking. While vendors can compile their programs using static sanitizers like AddressSanitizer to monitor all code within Linux-based firmware, this approach is not feasible for third-party security analysts. Analysts typically work with binaries, not source code, to uncover vulnerabilities. But the binary diversity issues of file formats, architectures, symbol tables, and relocation information may result in unexpected runtime errors in the instrumented program, and static sanitizers hardly detect flaws beyond the fuzz-input program, especially those occurring within Dynamic Link Libraries (libraries). In contrast, dynamic sanitizers [5, 11, 12] inject detection codes at runtime through dynamic binary instrumentation (DBI) and monitor process executions, rather than rewriting the binaries. Hence, the dynamic sanitizers prevent the binary diversity issues and extend the detection surface.

The proliferation of Internet of Things (IoT) devices has given rise to a plethora of security concerns, necessitating the development of various fuzzers (e.g., FirmAE [1], FirmFuzz [13], and FirmAFL [14]) and memory sanitizers (e.g., AddressSanitizer [6], MemorySanitizer [15], and ThreadSanitizer [16]).

The function of fuzzers is to identify flaws by sending test cases to devices and monitoring for abnormal behavior, while memory sanitizers enhance these tools by improving the detection of memory-related vulnerabilities. However, these sanitizers are designed for source code, whereas many commercially available IoT devices use closed-source firmware and therefore can only access the binaries.Some sanitizers can work for embedded binaries without an operating system. For instance, µSBS [8] is capable of analysing bare-metal firmware (i.e. firmware that runs directly on hardware) by employing static rewriting techniques to expose memory corruption issues.Furthermore, some sanitizers [5, 11] are suitable for firmware with systems, especially Linux-based firmware, which is widely utilised in IoT devices.These firmware provide essential services like network management and support various hardware architectures, including ARM and MIPS. However, existing sanitizers for Linux-based firmware still face significant challenges in terms of detection capability, coverage, and overhead.

Challenge. With meticulous analysis, we note that designing a memory sanitizer for Linux-based firmware confronts with three main challenges. ***(C1)*** Many memory vulnerabilities in the Linux-based firmware only trigger *Silent Memory Corruptions* (*Silent-MCs*). Silent-MCs refer to potential memory corruptions that do not result in externally observable abnormal program behaviors, thus remaining "silent" to both users and fuzzers [17]. Embedded Linux systems, commonly used in Linux-based firmware, differ significantly from traditional desktop systems. They often lack robust memory protection mechanisms and operate with limited performance capabilities. These limitations not only result in a higher incidence of silent memory corruptions but also make detecting such issues exceptionally challenging. ***(C2)*** Linux-based firmware comprises multiple interactive service programs and vendor-customized libraries. These programs and libraries may contain a large number of vulnerabilities due to vendors' poor design or lack of security considerations [18, 19]. Most existing binary sanitizers are designed for executables, including the main programs that serve as the core backend service programs handling requests [15, 8, 9]. However, they generally do not detect dynamically loaded code in libraries because they are considered trustworthy, which limits their effectiveness in detecting vulnerabilities across the broader Linux-based firmware. ***(C3)*** Global memory monitoring usually results in a huge performance overhead, especially for complexly structured and environment-restricted Linux-based firmware. Existing sanitizers use techniques such as *redzone*, *shadow memory*, etc., to implement runtime tracking and logging of each memory access operation, with additional checking and processing operations before and after accessing each memory location [10]. These operations result in excessive computational and storage overhead, which would significantly slow down the firmware or even make it stagnate.

Solution. To address these challenges, we present DFirmSan, a dynamic memory sanitizer for Linux-based firmware that can be integrated into existing fuzzers. DFirmSan introduces a key component, *dynamic boundary checking* (DBC), to detect memory corruptions in service programs and vendor-customized libraries. DBC identifies all relevant binaries in the firmware pack-

age and applies dynamic binary instrumentation (DBI) to monitor them during runtime. By tracking data discrepancies outside memory boundaries, DBC can reveal memory corruption that occurs beyond the allocated space. DFirmSan prioritizes monitoring critical memory spaces associated with memory-sensitive functions (such as standard library functions that modify specific memory areas), which helps minimize overhead. Based on our analysis of real-world memory vulnerabilities, we observe that most memory vulnerabilities in Linux-based firmware are linked to these sensitive functions, with few exceptions like loop-overflow issues. Previous studies support this observation [20, 21, 22, 23]. To enhance the efficiency of DFirmSan, we employ pre-analysis of sensitive info to examine target binaries and generate a detailed report. This report gathers holistic information on sensitive function call addresses and destination parameters, enabling DBC to accurately identify detection points and monitor destination parameters. Additionally, we introduce a *untrusted data tracking* approach to remove redundant detection points. By focusing dynamic memory monitoring on sensitive function parameters that may be influenced by input data, rather than all variables, this method significantly reduces overhead and supports lightweight memory corruption detection. To further reduce false positives, we propose a *dynamic boundary update* mechanism. When an anomaly is detected, DFirmSan evaluates candidate false positives by considering the function type, variable type, and boundary markers. Based on this analysis, the boundaries of critical parameters are updated—either merged or expanded—into definitive boundaries. These updates are then reflected in the pre-analysis report, ensuring that the same false positive will not recur at that specific detection point.

Performance. We implement a prototype of DFirmSan and conduct a systematic evaluation on 18 real-world Linux-based firmware samples from 12 well-known IoT vendors. The results show that DFirmSan, when incorporated into two advanced fuzzers, finds 117 and 25 additional previously reported CVEs, respectively, while introducing only 16.43% and 2.69% additional time overhead. In a broader vulnerability mining of Linux-based firmware, DFirmSan demonstrates its scalability and helps uncover several new zero-day vulnerabilities, including 4 CNVD vulnerabilities. To assess the effectiveness, we then select 21 CVE vulnerabilities covering the majority of memory vulnerabilities across different firmware and analyze the number of proof-of-concepts (PoCs) generated by DFirmSan. As a result, DFirmSan increases the number of PoCs from 62 to 3,408, with only 12 false positives (0.35%). Validation confirms that over 97% of the PoCs successfully trigger *Silent-MCs*, highlighting DFirmSan's capability to detect subtle and hard-to-capture vulnerabilities.

Contributions. Our main contributions are threefold:

- In this paper, we introduce DFirmSan, a lightweight dynamic binary memory sanitizer specifically designed for embedded Linux-based firmware. This innovative tool enables the detection of *silent memory corruption* resulting from memory vulnerabilities.

- We propose several novel methods to enhance the detection capability and

performance of DFirmSan. The *dynamic boundary checking* technique is specifically designed to identify memory corruptions, particularly Silent-MCs, within service programs and vendor-customized libraries in Linux-based firmware. To reduce additional overhead, we introduce the *pre-analysis of sensitive info*, which enables DBC to focus on the necessary memory spaces related to sensitive functions. Additionally, our *dynamic boundary update* approach effectively reduces false positives, contributing to the overall accuracy and efficiency of DFirmSan.

- We systematically evaluate DFirmSan's effectiveness and practicality on 18 real-world Linux-based firmware. The results indicate that DFirmSan assists two advanced fuzzers in detecting 117 and 25 additional known CVEs, all while maintaining low overhead and a minimal false positive rate. In a more extensive range of firmware testing, DFirmSan demonstrates scalability and helps uncover several zero-day vulnerabilities, including 4 CNVD vulnerabilities. We have released the code and the dataset to ease the follow-up research: https://github.com/dierye/dfirmsan.

2. Related Work

2.1. Sanitizer

Sanitizer is a technique employed to identify software flaws, typically used alongside fuzzers. It offers detection mechanisms during program testing to uncover memory safety issues, such as buffer overflows and null-pointer dereferences. Table 1 illustrates the techniques utilized by existing state-of-the-art sanitizers, comparing their time overheads with that of the original program running without any sanitizer. Sanitizers can be categorized into two types based on their instrumentation techniques: static sanitizers and dynamic sanitizers.

Static sanitizers instrument the source code or binary, providing memory detection capabilities and generating modified programs that can be executed directly. However, their functionality is often limited to specific platforms. One of the most well-known static sanitizers is AddressSanitizer (ASan) [6], which employs red zones and shadow memory to inspect the memory of stack objects, global variables, and dynamic heap objects. ASan also replaces standard library functions with error-reporting functions.ASan effectively detects memory errors in x86 programs, typically introducing only $1-3\times$ the memory and execution overhead compared to the original program. However, due to security and commercial constraints, the source code is frequently unavailable, prompting researchers to focus on binary sanitizers for closed-source applications. An example is RetroWrite [7], which implements a static binary rewriting technique that achieves similar functionality to ASan, albeit with higher memory overhead. Some static sanitizers designed for ARM architecture binaries leverage hardware or instruction features to achieve exceptionally low overhead. For instance, MTSan [4] employs Pointer Tagging and Memory Tagging techniques to detect memory safety violations in ARM binaries, achieving overhead comparable to source instrumentation (1-3×). Additionally, ARMore [24] focuses on

combining layout replication and rebound tables to restore code pointers, while utilizing hardware assistance to reliably detect and adjust data access within code sections. Furthermore, some sanitizers target bare-metal firmware, such as µSBS [8], which employs static rewriting techniques to make memory corruption faults observable on devices. However, these sanitizers are generally suited for relatively simple, standalone programs. Their reliance on static methods limits their ability to detect dynamic runtime flaws, including vulnerabilities introduced by vendor-customized libraries. Although some of them can instrument all external libraries to broaden the detection scope, they face more significant challenges, particularly in compatibility, due to the size and complexity of these libraries.

Table 1: Comparison of Existing Sanitizers.

Sanitizer	Instrumentation Method	Target	Bug-finding Techniques	Library Detection	Runtime Overhead
AddressSanitizer	Source-Code Instrumentation	Source-Code	①, ②	Y	1-3×
Retrowrite	Binary Rewriting	x86 Binary	①, ②	N	1-3×
ARMore	Binary Rewriting	Arm Binary	①, ②	Y	1-3×
MTSan	Binary Rewriting	Arm Binary	③, ④	N	1-3×
µSBS	Binary Rewriting	Bare-Metal Firmware	①, ②	N	1-3×
Memcheck	DBI	Multi-Arch Binary	①, ②, ⑤	Y	>10×
QASan	DBI	Multi-Arch Binary	①, ②	Y	>10×
DFirmSan	DBI	Linux-Based Firmware	⑥, ⑦	Y	1-3×

Bug-finding Techniques: ① Redzone, ② Shadow Memory, ③ Pointer Tagging, ④ Memory Tagging, ⑤ Uninit Value Tracking, ⑥ Sensitive Info Analysis, ⑦ Dynamic Boundary Checking

Dynamic sanitizers utilize dynamic binary instrumentation (DBI) during program execution, enabling more thorough memory detection, albeit with high overhead costs. Memcheck [5] monitors memory allocation and deallocation in x86 applications to identify memory errors. It is a robust and powerful dynamic binary analysis tool but incurs significant performance penalties. Similarly, QASan [11] facilitates the detection of heap memory violations across various architectures. By re-hosting binaries on the QEMU translation framework [25], QASan also incurs substantial overhead. Research indicates that these dynamic sanitizers typically impose a time overhead exceeding 10× and a memory overhead ranging from 3 to 10× [4]. Such resource demands render them impractical for fuzzing scenarios, especially when applied to firmware, which have to be operated under strict performance constraints.

Consequently, there is currently no low-overhead sanitizer that effectively supports comprehensive memory monitoring for Linux-based firmware. In response to this gap, we propose DFirmSan, a lightweight dynamic memory sanitizer designed to maintain the low overhead (1-3× for time) essential for firmware fuzzers while enhancing the detection of memory corruption.

2.2. Firmware Fuzzer

Traditional fuzzers primarily concentrate on optimizing fuzzing algorithms and techniques, whereas firmware fuzzers emphasize creating a suitable testing environment. These fuzzers can be divided into two categories: bare-metal firmware, which interacts directly with hardware, and embedded firmware, which utilizes an embedded operating system or real-time operating system (RTOS) to manage hardware resources and provide a platform for application execution. It is important to note that most embedded firmware fuzzers specifically target Linux-based firmware, reflecting the widespread use of embedded Linux systems in IoT devices.

Bare-metal firmware fuzzers concentrate on delivering precise high-level emulation and generating random inputs via appropriate hardware interfaces. For instance, P^{2}IM [26] employs approximate emulation to provide suitable inputs to the firmware, achieving a sufficiently coarse emulator for effective firmware fuzzing while maintaining low overhead. Similarly, several other studies [27, 28, 2, 29] focus on optimizing firmware re-hosting to enhance fuzzing throughput.

Linux-based firmware fuzzers aim to encompass a broader range of real-world firmware and enhance fuzzing efficiency. Some fuzzers test directly on real devices, which yields the most accurate results [30, 31, 32]. However, this approach can be inefficient and may risk damaging the devices. To address these challenges, Zheng et al. developed FirmAFL [14], which emulates 8,556 Linux-based firmware images using Firmadyne [33] and boosts fuzzing speed by integrating system-mode and user-mode emulation with QEMU [25]. Many subsequent firmware fuzzers [1, 13, 34] are fundamentally built on the QEMU framework and focus on enhancing fuzzing capabilities.

2.3. Fault Monitor

Due to the absence of memory safety mechanisms in firmware, many program flaws resulting from memory corruption remain undetectable externally. As a result, firmware fuzzers that depend on crash signals must incorporate specialized crash detection modules, referred to as fault monitors. These methods can be classified into two categories: *external probing* and *internal checking*.

External probing monitors the operational status of firmware through messages or logs generated during execution. For instance, IoTFuzzer [30] sends heartbeat messages to firmware programs and awaits a response; however, this method can suffer from delayed feedback due to unstable communication processes. Similarly, FirmFuzz [13] employs Helper Injection to identify various vulnerabilities, including command injection, by analyzing system call logs. Nevertheless, this approach still depends on the exception handling mechanisms of the system kernel.

Internal checking detects faults by integrating crash-trigger mechanisms into firmware or emulators. For example, FirmWire [35] injects a FUZZ-task into a running firmware image to induce specific crashes. However, this method assumes prior knowledge of the vulnerability's details, limiting its generalizability. P^{2}IM [26] utilizes an emulator that implements basic memory error detection

by assigning minimum permissions for each memory segment, treating any access violations as indications of memory corruption. Nonetheless, determining the scope and permissions of individual memory segments can be complex in intricate firmware, leading to coarse-grained segment detection. FirmCorn [22] enhances crash detection by analyzing stack data following calls to sensitive functions and monitoring program exceptions. While this approach demonstrates strong detection capabilities, it requires extensive memory monitoring, resulting in significant additional overhead. Additionally, its reliance on the specific implementation of the fuzzer limits its general applicability.

3. Silent Memory Corruption

Fuzzing relies on detecting observable program faults, such as exceptions, crashes, or incorrect outputs resulting from vulnerabilities. In desktop systems, fuzzers can easily identify these faults due to various memory protection mechanisms, including memory isolation, integrity checks, and StackGuard. These mechanisms enhance the visibility of memory corruptions and generate recognizable crash signals, such as SIGSEGV and SIGUSR, which fuzzers can effectively utilize.

However, most embedded systems in IoT devices lack effective memory protection mechanisms due to hardware constraints and performance considerations. Additionally, firmware programs are often designed to be resilient, enabling them to manage minor errors to ensure long-term device operation. Research by Muench et al. [17] indicates that the memory fault detection rate in embedded systems is 63% lower than in desktop systems. This implies that many memory corruptions within these devices go undetected from external sources, a phenomenon we refer to as Silent Memory Corruptions (Silent-MCs).

Listing 1: Code Fragment of Netgear R6400.

```
1  v31 = recvfrom(dword_C1248, data, 0x1FFFu, 0, &v60, &v63);
           //untrusted data entrance
2  /* ... */
3  memset(buf, 0, 64u); //buf is 64 bytes in the heap
4  /* ... */
5  char *__fastcall sub_D524(char* data, char *needle, const
       char *a3, char *buf)
6  {
7      /* ... */
8      if (!v8)
9      {
10         while (1)
11         {
12             /* ... */
13             v4 = strstr((const char *)data, v10); //data->v4
14             v14 = &v4[v11]; //v4->v14
15         }
16         /* ... */
```

```
17          if ( v15 - v14 < 1024 )
18              strncpy(buf, v14, v15 - v14); //v14->buf
19      }
20  }
```

We illustrate the concept of Silent-MC using the Buffer Overflow (BOF) vulnerability **CVE-2021-34991** found in the Linux-based firmware *Netgear R6400*. As show in Listing 1, the vulnerability begins when untrusted input is received via the *recvfrom* function, which allows up to 0x1FFu (8191) bytes to be stored in the variable *data*. Later, in the subroutine *sub_D524*, this input is processed using *strncpy* to copy untrusted data into *buf*, which has a maximum length of only 64 bytes. The code checks whether the length to be copied is less than 1024 bytes, failing to consider the actual size of *buf*. Consequently, if the copied data exceeds 64 bytes, a buffer overflow occurs, potentially leading to memory corruption, as illustrated in Figure 1. Due to the absence of memory protection mechanisms in the firmware's operating environment, the heap space beyond the *buf* boundary may consist of unused or unallocated memory. This situation can lead to unobservable memory errors, as overwriting this area may not immediately trigger any detectable issues. This scenario exemplifies Silent-MC, where the buffer overflow remains unnoticed, resulting in potential long-term instability or vulnerabilities within the system without any immediate failure signals.

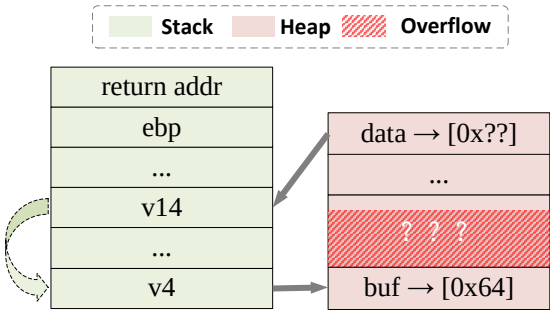

Figure 1: Memory Space Analysis of CVE-2021-34991.

Some fuzzers have developed specific strategies to address the challenge of detecting Silent-MCs in firmware [13, 34, 35, 36]. These strategies include utilizing a heavy memory monitoring system for emulated firmware or intentionally causing program crashes by generating overflow data to overwrite function return addresses. However, these approaches are not always practical. For instance, an overloaded memory monitor can hinder the performance of both the firmware and the fuzzers. Additionally, the complexity of memory spaces and firmware architectures complicates efforts to induce program crashes artificially. As a result, current methods for detecting Silent-MCs in firmware exhibit notable limitations in terms of both general applicability and efficiency.

In this paper, we propose a method for detecting Silent Memory Corruption

(Silent-MC) grounded in the observation that the C/C++ standard library includes a set of functions—referred to as *sensitive functions*—that are prone to memory vulnerabilities when key parameters can be influenced by the user input. Table 2 provides examples of typical memory-sensitive functions. In fact, any functions with memory write operations can be regarded as memory-sensitive functions, such as *memcpy* and *memmove*.

Table 2: Sensitive Functions.

Function Name	Function Prototype	Function Type (Source->Destination)
gets	char *gets(char *dest)	I : N → 1
strcpy	char *strcpy(char *dest, const char *src)	
strncpy	char *strncpy(char *dest, const char *src, size_t n)	
strcat	char* strcat(char* dest, char* src)	
sprintf	int sprintf(char *dest, const char *format [, src1,...])	
vsprintf	int vsprintf(char *dest, const char *format, va_list src)	
fread	size_t fread(void *dest, size_t size, size_t count, FILE *src)	
scanf	int scanf(const char *format,T *dest1, ...)	II : 1 → N
sscanf	int sscanf(const char *src, const char *format,T *dest1, ...)	
fscanf	int fscanf(FILE *src, const char *format,T *dest1, ...)	
vfscanf	int vfscanf(FILE *src, const char *format, va_list dest)	
vscanf	int vscanf(const char *format, va_list dest);	
vsscanf	int vsscanf(const char *src, const char *format, va_list dest)	

Our key insight is that any memory modification occurring beyond the boundaries of the destination parameter after a sensitive function call is considered abnormal. In other words, if out-of-bounds data is altered following the execution of a sensitive function, it indicates that memory corruption has taken place.

4. DFirmSan

4.1. Overview

Figure 2 provides an overview of our approach. The primary objective of DFirmSan is to detect Silent Memory Corruptions (Silent-MCs) by monitoring the boundary data of sensitive variables at both the start and end of sensitive function calls. To achieve this, our solution comprises two key components: Pre-analysis of Sensitive Information and Dynamic Memory Corruption Detection.

Pre-analysis of Sensitive Info. This phase occurs while the firmware is inactive, allowing for the collection of essential information through static analysis. This information is then stored as a pre-analysis report to facilitate Dynamic Memory Corruption Detection. DFirmSan examines service programs and vendor-customized libraries within Linux-based firmware, identifying the holistic call addresses of sensitive functions as key detection points and extracting information about associated critical parameters. Since only those detection points influenced by external inputs can lead to memory vulnerabilities, we introduce the concept of *untrusted data tracking* to eliminate unnecessary detection points. This involves conducting a coarse-grained forward data flow

analysis that begins at binary data entry points to determine which detection points are affected by external inputs.

Dynamic Memory Corruption Detection. This phase occurs during the firmware's execution to identify memory vulnerabilities in real time. DFirmSan implements *dynamic boundary checkin* (DBC) by utilizing sensitive information from the pre-analysis report and dynamically instrumenting vulnerability detection points. When the program pointer reaches a detection point, the memory boundary addresses of critical parameters are calculated based on both sensitive information and dynamic execution data. Discrepancies to out-of-bounds data are then monitored before and after the sensitive function call to detect any memory corruption. Additionally, we address the issue of false positives in memory corruption detection by introducing *dynamic boundary update*. When DBC identifies a potential memory corruption, it suspects that critical parameters may be influenced by adjacent memory spaces. It also considers the possibility of incorrect boundary definitions due to errors in the inferred stack variable size from the disassembly engine. In response, the affected boundaries will be merged or expanded, and these changes will be updated in the pre-analysis report.

The Assumptions about Firmware Binaries. DFirmSan's effectiveness relies on pointers to calculate memory boundary addresses. Complex compiler optimizations can significantly complicate the data dependencies within firmware binaries, reducing the accuracy of static data flow analysis and limiting its applicability. Additionally, when firmware utilizes *position-independent code* (*PIC*) or *position-independent executables* (*PIE*), dynamic instrumentation encounters challenges in identifying precise vulnerability detection points, thereby restricting its effectiveness. For binaries compiled with the *-fomit-frame-pointer* option, the absence of a frame pointer complicates the inference of stack variable boundaries, making it difficult to analyze stack-based vulnerabilities. However, DFirmSan remains capable of detecting memory corruption on the heap, as heap variables are accessed directly through pointer addresses during runtime, which is unaffected by the omission of frame pointers. Importantly, DFirmSan does not rely on debugging information or symbol tables, allowing it to analyze stripped binaries effectively. This ensures broad applicability across a wide range of firmware binaries, including those lacking detailed symbolic information.

4.2. Pre-analysis of Sensitive Info

The objective of the *pre-analysis of sensitive info* is to identify and extract sensitive function details from service programs and vendor-customized libraries within the firmware, and then generate a comprehensive pre-analysis report. This process involves three key steps:

The first step is to identify service programs and vendor-customized libraries within Linux-based firmware. Devices typically offer various interactive services, such as web servers, Telnet/SSH, and FTP. To achieve this, we leveraged the technique of the firmware static analysis tool Karonte [37], which identifies border programs that export network services and broadens the analysis scope

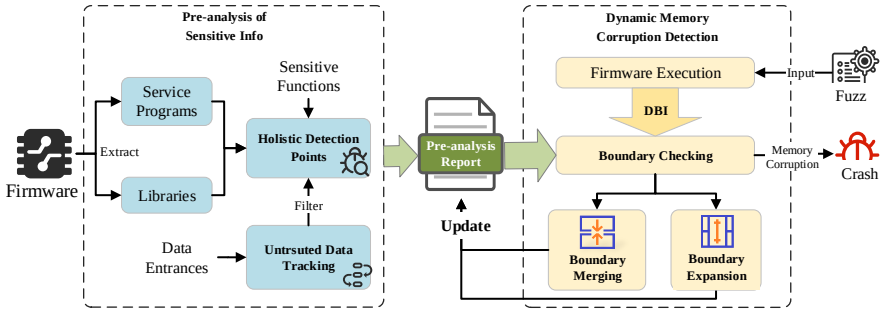

Figure 2: Workflow of DFirmSan.

by tracing data flows between binaries. Additionally, we introduced two complementary methods to enhance service program identification: 1) searching for initialization scripts and configuration files in the firmware root filesystem directory to analyze the service programs they launch; and 2) searching for string fields associated with bound service ports in the firmware and reverse-engineering their corresponding service processes. Next, we utilize the ELF viewer tools and the binary analysis tools to extract all libraries associated with the identified service programs. To automatically identify vendor-customized libraries among these, we have established three criteria: 1) file names, header information, and function names that include vendor-specific keywords; 2) libraries that do not belong to standard C/C++ libraries; and 3) libraries that cannot be found on GitHub. A library is classified as a vendor-customized library if it meets Rule 1 or satisfies both Rule 2 and Rule 3.

The second step involves identifying holistic sensitive function calls as candidate detection points through the analysis of target binaries (i.e., service programs and vendor-customized libraries). Sensitive functions refer to library functions that modify memory data and accept user-controlled parameters. When these functions are called, the *source* data may be influenced by external inputs. If the memory spaces of the *destination* parameters are not strictly managed, this can lead to memory vulnerabilities such as out-of-bounds writes and buffer overflows. The mapping relationship between source and destination parameters can lead to different outcomes in terms of memory corruption. Therefore, we classify sensitive functions into two categories, as detailed in Table 2: (I) N→1, where data from multiple sources is formatted and written to a single destination address (e.g., *sprintf()*); and (II) 1→N, where data from a single source is parsed and written to multiple destination addresses (e.g., *sscanf()*). Functions with a 1→1 relationship are categorized as Type I. Compared to simpler functions, those exhibiting an N→N relationship are more complex. However, these functions are implemented using both Type I and Type II sensitive functions, so there is no need for separate consideration.

We utilize a disassembly engine to locate the call addresses of sensitive functions within target binaries and identify their destination parameters. It is important to recognize that alias analysis and indirect call refining are common

challenges in binary analysis. Existing research [38, 39] has made significant progress in addressing these challenges, and although some limitations remain, these methods considerably reduce the effort required. This allows us to focus on the specific challenges we face. At each sensitive function call site, we identify the destination parameters and gather their addresses, space sizes, stack-/heap information, and details about neighboring variables. It is important to note that real addresses for these parameters cannot be determined during the pre-analysis phase, as variable addresses vary depending on the firmware's operation. Therefore, we identify the corresponding registers or stack offsets for the parameters, with the actual addresses to be resolved later during DBC.

In addition, the method for determining the memory space size of the target parameter differs based on whether the variable is located on the heap or the stack. For heap variables, which are typically allocated explicitly (e.g., through the *malloc* function), the space size can be directly obtained by analyzing the relevant memory allocation code. In contrast, for stack variables, we determine the memory space size by analyzing the data type and stack frame layout using the IDA Reverse Engineering Framework. However, due to compiler optimizations or complex pointer operations in target binaries, the inferred sizes of some stack variables may be inaccurate. This can lead to false positives, which is why we introduce the *dynamic boundary update* mechanism in § 4.3 to mitigate this issue.

The third step is to minimize redundant detection points, thereby reducing unnecessary overhead. Many IoT devices are designed for efficiency, often at the cost of security. As a result, sensitive functions are frequently misused in various firmware implementations, increasing the risk of memory errors. For instance, we identified 1007 and 269 detection points in the *httpd* service of the Tenda AC18 and the *upnpd* service of the Netgear R6400, respectively. However, many of these points are unrelated to external inputs. The sensitive functions at these points do not receive untrusted data as parameters and are therefore not vulnerable, making them redundant. To address this, we propose an *untrusted data tracking* method to filter out redundant detection points. This method performs coarse-grained forward data flow analysis starting from the program's untrusted data entrances, identifying sensitive function calls that could potentially be affected by untrusted data.

For target binaries in firmware, untrusted data entrances can be categorized into five types: **1) Network sockets**, which receive data from connected sockets, commonly found in web service programs; **2) File entrances**, which read data from files; **3) Parameter entrances**, where programs and library functions receive input from users or processes; **4) Implicit receptions**, which read data from global variables or shared memory; and **5) Hardware entrances**, which receive peripheral data through interfaces like GPIO and UART. We identify these data entrances based on distinct function prototypes and mark their untrusted parameters. For instance, in line 1 of Listing 1, the service program *upnpd* in the Netgear R6400 firmware receives network data through the *recvfrom* function, marking the *input* parameter as untrusted.

Next, we employ a forward data flow analysis beginning at untrusted data

entry points, intentionally adopting a coarse-grained approach to simplify the process. First, jump conditions introduced by conditional statements or loops are ignored, and we assume all branches are executed without performing precise path analysis. Additionally, data sanitization steps in the code, such as input validation and data type conversion, could filter or transform untrusted data into a harmless form. However, checking the impact of these steps on untrusted data across all execution paths is complex and time-consuming; therefore, they are not considered here. Instead, we mark detection points that may be influenced by untrusted data, deferring the evaluation of path feasibility and data sanitization to the dynamic detection phase. In this phase, whether a detection point is reached and whether data is sanitized is verified through runtime checks. For external library function calls, our treatment depends on their origin. If the function is part of a vendor-customized library, we analyze the library file to trace the propagation of untrusted data. Conversely, standard library functions are treated as black boxes. In such cases, the effect on parameters or return values is determined solely based on predefined annotations, without analyzing the library's internal logic. This coarse-grained approach prioritizes simplicity and broad coverage over precision, sacrificing fine-grained tracking of untrusted data propagation. It avoids detailed path exploration, ignores sanitization efforts, and relies on annotations or assumptions for external functions. However, by ensuring that the remaining detection points after this process encompass all potentially vulnerable sensitive function call points—even if redundancies remain—we maintain comprehensive vulnerability coverage without compromising detection quality.

Furthermore, for unknown library functions, we adopt an *Over-Untrusted* strategy, assuming all return values and pointer parameters are affected. While this approach slightly impacts detection efficiency, it enhances the likelihood of uncovering more valuable vulnerabilities at over-untrusted detection points. For instance, in Listing 1, the variable *data* causes *v4*, *v14*, and *buf* to be marked as untrusted on lines 11, 12, and 16, without needing to consider conditional statements, loops, or the internals of library functions.

After untrusted data tracking, DFirmSan remove candidate detection points that are not on the compromised path to avoid unnecessary monitoring of related parameters.

Before proceeding to the dynamic testing phase, the sensitive information gathered from the previous steps is consolidated and stored in the pre-analysis report. Figure 3 demonstrates how DFirmSan aids in detecting Silent-MCs triggered by CVE-2021-34991. During pre-analysis, DFirmSan identifies service programs and vendor-customized libraries such as *upnpd* and *libupnp.so* from the firmware of the Netgear R6400. When locating vulnerability detection points, the function call *stnpcy* at address 0xD5F4 is selected as a candidate. Additionally, all untrusted data entrances within the binary are identified, including functions like *get_cgi()* and *recvfrom()*. Through untrusted data tracking, DFirmSan detects a program path where data introduced by the *recvfrom()* function at address 0x1BD28 propagates to the candidate detection point at 0xD5F4. Therefore, the relevant information for this detection point is then

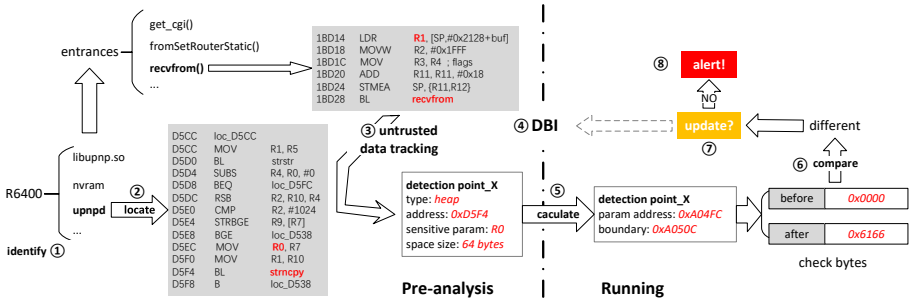

Figure 3: A Case for Using DFirmSan to Detect Silent-MC in Netgear R6400.

recorded in the pre-analysis report.

4.3. Dynamic Memory Corruption Detection

In the Dynamic Memory Corruption Detection phase, DFirmSan identifies memory flaws during sensitive function calling. At firmware initialization, boundary check functions and the pre-analysis report are loaded into an allocated memory space. When the program pointer reaches a sensitive function call, DFirmSan checks if it corresponds to a detection point. For detection points, two boundary check functions are invoked—one before and one after the sensitive function call.

Before Calling Sensitive Function. The first step involves determining the memory addresses and boundaries of destination parameters by integrating information from the static pre-analysis report with dynamic execution data. If the parameter address or pointer is stored in a register, the actual address can be directly accessed at the detection point. However, if the parameter is accessed through offset, then the real address needs to be calculated using following equation.

$$mem\_addr = func\_start + offset\_addr \tag{1}$$

Where *func_start* is the base address of the function stack frame, *offset_addr* is the offset of the parameter, and *mem_addr* is the real address. After getting the real address, the boundary of the parameter can be calculated by the following equation.

$$boundary\_addr = mem\_addr + param\_size \tag{2}$$

Where *param_size* is the space size of the parameter, and *boundary_addr* is the boundary address of the parameter. DFirmSan then reads 2 or 4 bytes of memory data beyond the boundary addresses of the destination parameters and logs this data as *check bytes*.

After Calling Sensitive Function. DFirmSan re-reads the *check bytes* from memory and compares them to previously recorded values. Discrepancies in *check bytes* indicate possible data tampering beyond the boundaries of the

destination parameter at the detection point. However, such discrepancies do not definitively signal memory corruption, as they could also result from adjacent memory space interference or erroneous boundary inferences, leading to potential false positives.

Dynamic checking's false positives arise from two sources. First, Effects from adjacent memory spaces can occur when a sensitive function has multiple destination parameters. If neighboring parameters are modified simultaneously, it may result in discrepancies to one of the *check bytes*. For example, Figure 4 illustrates the stack frame associated with vulnerability CVE-2022-38310, as shown in Listing 2. In this case, simultaneous modifications to the parameters s1 and v9 can inadvertently affect the check bytes between them, even though this does not indicate memory corruption. More broadly, when these destination parameters are member variables of a class object, their memory allocations are often contiguous. Second, erroneous boundaries may result from inaccurate memory size estimates during the binary pre-analysis phase, leading to incorrect boundary calculations. Applying DBC on these flawed boundaries can yield misleading results.

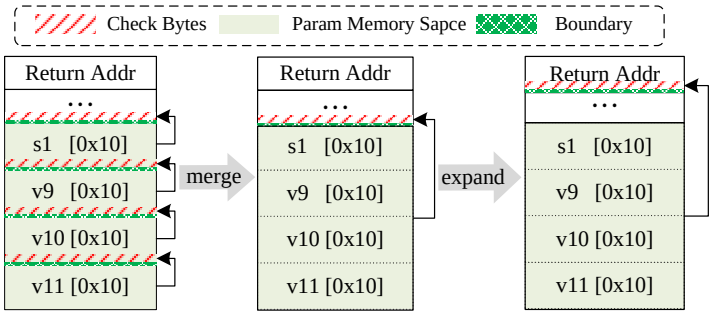

Figure 4: Dynamic Boundary Update.

Erroneous boundaries may result from inaccurate memory size estimates during the binary pre-analysis phase, leading to incorrect boundary calculations. Applying DBC on these flawed boundaries can yield misleading results. We conduct a thorough analysis, which serves as a foundation for developing strategies to mitigate these false positives. Table 3 provides a summary that outlines the memory locations of parameters, the types of sensitive functions involved, and the underlying causes of false positives. False positives occur in four primary scenarios:

1. Adjacent memory spaces of multiple stack destination parameters for Type-II sensitive functions;
2. Erroneous Boundaries of stack destination parameters for Type-I sensitive functions;
3. Erroneous Boundaries of multiple stack destination parameters for Type-II sensitive functions;

4. Adjacent memory spaces of multiple heap destination parameters for Type-II sensitive functions.

Notably, parameters stored in heap memory, due to their well-defined boundaries, do not typically result in false positives except in the case described as Case 4.

Table 3: Analyze Scenarios of False Positives.

Memory Location	Function Type	Reason	False Positive
Stack	II	Adjacent memory space impact	Y
	I	Error boundary	Y
	II	Error boundary	Y
Heap	II	Adjacent memory space impact	Y
Heap	I	Definite boundary	N
	II	Definite boundary	N
Stack/Heap	**I/II**	**After Dynamic Boundary Update**	N

Listing 2: Code Snippets of CVE-2022-38310

```
1 int sub_79180(const char *a1, char *list, unsigned int a3){
2     char s1[16], v9[16], v10[16], v11[16];
3     char *src= list;
4     /* ... */
5     if ( sscanf(src, ``%[^,],%[^,],%[^,],%s'', v11, v10, v9
          , s1) == 4 )  // vulnerability
6     /* ...*/
7 }
```

To minimize false positives, we have developed a Dynamic Boundary Update method. When dynamic checking detects alerts for boundary discrepancies, DFirmSan handle false-positive scenarios by performing boundary merging and expansion. Alerts associated with updated boundaries are flagged as candidate false positives for further evaluation. Specifically: 1) For multiple parameters residing in adjacent memory spaces, we "merge" their memory areas into a single entity that shares the boundary and *check bytes* of the outermost parameter. This approach prevents modifications of these parameters from affecting the *check bytes*. As demonstrated in Figure 4, the parameters *s1* and *v9* are located in adjacent memory spaces and are the destination parameters of the same *sscanf()* call. Consequently, when *s1* is altered, it is incorrectly identified as *v9* overflow, resulting in a false positive. Therefore, these parameters will be merged, sharing the boundary and *check bytes* of *s1*. 2) For stack destination parameters with potentially incorrect boundaries, we expand the boundary to the starting address of the parent function's stack frame header—typically where the return address is stored—and extract the *check bytes* from that location. Since the function's stack frame header remains fixed until the function exits, any discrepancies in the *check bytes* accurately indicate memory corruption. As illustrated in Figure 4, the variable *s1* is a stack variable whose boundary

may be erroneously inferred by the reverse engine during the pre-analysis phase. Dynamic checking of erroneous boundaries can lead to false positives. Therefore, its boundary is extended to the start of the function stack frame.

When the firmware exits, the updated boundary information is synchronized with the pre-analysis report. Subsequent test processes will validate both the initial boundaries from the analysis report and the updated boundaries. If a test case triggers boundary violations at both the initial and updated boundaries at that detection point, it confirms a genuine vulnerability rather than a false positive. In such cases, all candidate alerts related to this detection point are accepted as true; otherwise, they are excluded from the test results after all tests are complete. By specially handling false positives, this approach can reduce noise in the fuzzing results while remaining vigilant for potential vulnerabilities that may be overlooked.

It is important to note that the memory gap between the updated boundary and the parameter address can complicate the triggering of discrepancies in the *check bytes*. This challenge can be addressed during the fuzzing process. After completing a single fuzzing session and performing dynamic boundary updates, DFirmSan sends a special signal, termed *FP_SIGNAL*. When the fuzzer receives the *FP_SIGNAL*, it adds the test case that currently triggers a false positive to the fuzz seed priority queue. In subsequent fuzzing sessions, this test case will undergo additional mutations. If a genuine vulnerability exists, the discrepancies in the *check bytes* at the updated boundary will again trigger a report of memory corruption.

Algorithm 1 illustrates the full process of boundary checking after a call of sensitive function. The input, labeled as sensInfo, represents the details associated with the current detection point for the sensitive function. This collection of data includes the following components: (1) addr: the memory address of the detection point; (2) bound: the boundary of sensitive variable; (3) type: the classification of the sensitive function, denoted as I or II; (4) memLoc: the memory location of the sensitive parameter, indicating whether it resides on the stack or heap; (5) adjacentInfo: information regarding adjacent variables; (6) updateFlag: an indicator of whether the boundaries of the current detection point have been updated, initially false; (6) acceptFlag: an andicator of whether the boundary check alert for the current detection point is accepted, initially false.

DFirmSan begins by reading the *check bytes* data at the conclusion of the sensitive function call and comparing it with the previous record. If discrepancies are found, the process continues with further checks (lines 3-4). Next, if the current detection point is not a scenario where FP might occur, or if the alert at this detection point is acceptable (having passed boundary updates and checks), a CRASH_SIGNAL will be sent to report a crash (lines 5-6). If neither condition is met, a new alert will be created. If the detection point has not undergone a boundary update, two types of boundary updates will be performed as needed: boundary merging and boundary expansion. Following that, a candidate tag will be added to the alert, and an FP_SIGNAL will be sent, temporarily storing the candidate alert (lines 8-16). Otherwise, the check

Algorithm 1: Check Boundary Bytes and Update Boundary.

```
1  Function checkAndUpdate(sensInfo):
2      Inputs: sensInfo = { addr, bound, type, memLoc, adjacentInfo,
         updateFlag, acceptFlag};
3      curBytes ← readCheckBytes(sensInfo.bound);
4      if isDifferent(curBytes, preBytes[sensInfo.addr]) then
5          if sensInfo not in FP Scenarios or sensInfo.acceptFlag is true
             then
6              report(CRASH_SIGNAL);
7          else
8              alert ← create();
9              if sensInfo.updateFlag is false then
10                 sensInfo.updateFlag ← true;
11                 if isAdjacentMemory(sensInfo) then
12                     mergeBoundary(sensInfo);
13                 if sensInfo.memLoc is stack then
14                     expandBoundary(sensInfo);
15                 addCandidateAlerts(sensInfo.addr, alert);
16                 report(FP_SIGNAL);
17             else
18                 if CheckUpdatedBoundary(sensInfo) is true then
19                     acceptCandidateAlerts(sensInfo.addr, alert);
20                     sensInfo.acceptFlag ← true;
21                     report(CRASH_SIGNAL);
22                 else
23                     addCandidateAlerts(alert);
24                     report(FP_SIGNAL);
```

bytes of the updated boundary will be examined. If they have also changed, it indicates that there is indeed a vulnerability at this detection point. In this case, all relevant candidate alerts will be accepted, the acceptFlag will be set to true, and a CRASH_SIGNAL will be sent to report a crash (lines 18-21). If not, the current alert will continue to have a candidate tag added, and an FP_SIGNAL will be sent (lines 23-24). After all tests are completed, all temporarily stored candidate alerts that have not been accepted will be cleared.

Figure 3 illustrates each steps in detecting Silent-MCs triggered by CVE-2021-34991. The sensitive information gathered during the pre-analysis phase is dynamically instrumented into memory. When execution reaches the detection point, the actual memory address of the sensitive parameter (0xA04FC) is retrieved from register *R0*, and the corresponding memory boundary (0xA050C) is calculated. Data from the check bytes outside the boundary are read before

and after the *strncpy()* call. By comparing these values, DFirmSan identifies that the original value of 0x0000 has changed to 0x6166 (af). Given that this sensitive parameter is a heap variable with explicitly allocated memory, boundary updates are unnecessary. Consequently, an alert is triggered, indicating that memory corruption has occurred.

5. Implementation

We develop DFirmSan using approximately 1,100 lines of Python and 600 lines of C/C++ code. The pre-analysis module relies on IDA Pro [40] as its primary disassembly engine. To facilitate untrusted data tracking, we integrate several tools, including the multi-architecture binary analysis framework Angr [41], EmTaint [38] for alias analysis in firmware, and BPA [39] for refining binary indirect calls. This combination enhances DFirmSan's ability to analyze firmware effectively, ensuring robust detection of vulnerabilities related to untrusted data. To further expand our capabilities, we develop two sets of pre-analysis scripts tailored for ARM and MIPS architectures. This enables our approach to effectively address a wide range of Linux-based firmware architectures, demonstrating the versatility and generalizability of our solution.

Additionally, we integrate the dynamic sanitization module into QEMU [25], a widely used emulator core found in numerous advanced firmware fuzzers [34, 36, 42]. This integration enhances DFirmSan's functionality, enabling it to utilize the strengths of established tools and significantly improving its effectiveness in detecting vulnerabilities.

5.1. Pre-analysis

DFirmSan begins by identifying service programs and vendor-customized libraries within the firmware's root file system. It scans key directories, such as */etc/init.d/* and */etc/rc.d/*, to locate typical system initialization scripts or configuration files, including the *init.d/rcS* file, to identify all service programs launched during firmware initialization. Since some optional services may not be enabled by default, DFirmSan also examines associated web pages and applications to determine additional service names and their corresponding file directories. To enhance the identification process, DFirmSan incorporates advanced techniques from Karonte [37]. First, it detects binary files within the firmware that parse data received from network sockets, classifying these boundary binaries as service programs providing network services. It then constructs a binary dependency graph to identify additional service programs by analyzing data flow relationships between binaries. After pinpointing the service programs, DFirmSan uses the ELF viewer *readelf* [1] and the binary analysis tool *objdump* [2] to extract all libraries linked to these programs. Vendor-customized libraries are

[1] https://www.linux.org/docs/man1/readelf.html
[2] https://www.linux.org/docs/man1/objdump.html

isolated by excluding standard C/C++ libraries and system libraries, ensuring a focused analysis of custom components.

The next step involves identifying vulnerability detection points, a complex task. DFirmSan locates the call addresses of all sensitive functions within the disassembled code and determines the storage locations and memory sizes of the source and destination parameters based on the function prototypes. This process requires consideration of the different parameter-passing conventions in the ARM and MIPS instruction sets. Additionally, DFirmSan analyzes the data flow of parameter variables within the assembly code to ascertain their memory sizes. For heap variables, DFirmSan traces pointers and identifies functions such as *malloc* to determine memory sizes based on dynamic memory allocation. In contrast, for stack variables, DFirmSan examines assignment behaviors and pointer aliases back to their initial definitions, calculating the stack space size based on the function's stack frame, as inferred by IDA Pro.

Next, DFirmSan identifies all untrusted data entrances and conducts Untrusted Data Tracking to eliminate redundant detection points. During this process, it records the status of each non-standard function's parameter pointers and return values, marking them as untrusted. This prevents repeated analysis when the function is encountered again.

Finally, DFirmSan compiles the analysis results for all service programs and libraries, outputting them as a pre-analysis report in a specified format.

5.2. Dynamic Sanitization

The Dynamic Sanitization module implements dynamic instrumentation for memory boundary checking by modifying the QEMU source code. During firmware initialization, DFirmSan loads the boundary-checking function into a designated executable memory region for global access. It then invokes this checking function both before and after vulnerability detection points to execute tasks such as reading the pre-analysis report, calculating memory addresses, and comparing the *check bytes*. When memory corruption is confirmed and not deemed a false positive based on parameter type and neighboring memory space analysis, DFirmSan promptly exits the program and sends the *SIGSEGV* signal as a *CRASH* signal. In instances identified as false positives, DFirmSan adjusts the memory boundaries by either expanding or merging them, updates the pre-analysis report, and, at the conclusion of a fuzzing session, sends the *SIGUSR1* signal—a custom signal designated for user-defined purposes. Upon receiving the *SIGUSR1* signal, the fuzzing engine flags the current test case and adds it to the mutation queue, irrespective of whether it explores a new execution path.

It is important to highlight that the Dynamic Sanitization module is highly extensible and can be applied to various firmware emulation frameworks. For instance, when utilizing the Qiling emulation framework [43], its efficient hook interface allows us to implement a functionally identical dynamic sanitization module with fewer than 300 lines of hook code, minimizing the need for extensive modifications to QEMU. As a result, DFirmSan can support a diverse array of advanced firmware fuzzers with minimal migration effort required from security personnel.

6. Evaluation

We evaluate DFirmSan by considering the following research questions:

- **RQ1:** Can DFirmSan efficiently analyze and identify holistic sensitive detection points within firmware?
- **RQ2:** Can DFirmSan improve the efficiency of firmware fuzzers in discovering real-world vulnerabilities, and what is the time overhead involved?
- **RQ3:** Does DFirmSan enhance the capability to detect memory corruption, particularly Silent-MCs, and what is the associated false positive rate?
- **RQ4:** Can DFirmSan effectively scale across diverse firmware environments and help uncover zero-day vulnerabilities?

To answer these questions, we build a dataset comprising 18 Linux-based firmware samples based on three criteria: i) Is DFirmSan suitable for various vendors? ii) Is DFirmSan suitable for heterogeneous architectures? iii) Can DFirmSan detect different memory vulnerabilities? Based on these criteria, we first select 12 prominent IoT device vendors, including Tenda, ASUS, D-Link, H3C, TOTOLink, Netgear, TP-Link, Motorola, Linksys, Belkin, Trendnet, and Zyxel, as outlined in the report [44] Then, the two most prevalent IoT device architectures, ARM and MIPS, are selected according to the survey [45]. Finally, we select five of the most common memory vulnerabilities, including buffer overflow (CWE-119), stack overflow (CWE-121), heap overflow (CWE-122), format string vulnerabilities (CWE-134), and out-of-bounds writes (CWE-787). Detailed dataset information is provided in Table 4. We conduct extensive fuzzing on this dataset using two advanced firmware fuzzers, AFL++ [46] and FirmAE [1], and perform comparative experiments with related sanitizers, QASan [11] and Memcheck [5] First, we utilize DFirmSan's pre-analysis module to extract all service programs and vendor-customized libraries, assessing its effectiveness in identifying and filtering sensitive detection points (see § 6.1). Next, we conduct 24 hours of fuzzing across all firmware to document the number of discovered vulnerabilities (CVEs) and evaluate testing performance to assess DFirmSan's efficiency in discovering vulnerabilities in the real world. (see § 6.2). Then, we analyze the above fuzzing results, selecting a sample set of known firmware vulnerabilities to ensure broad coverage of memory-related issues. The number of CVE-related proof-of-concepts (PoCs) detected is compared before and after integrating DFirmSan and other sanitizers. Additionally, we evaluate the methodology's effectiveness and precision by analyzing the percentage of false positives and instances of Silent-MCs (see § 6.3). Finally, we evaluate the scalability of DFirmSan on a large-scale dataset provided by a SOTA research [47] and utilize DFirmSan to help uncover zero-day vulnerabilities in real-world firmware (see § 6.3).

Table 4: Memory Vulnerabilities in Firmware Samples.

Vendor	Model	Firmware Version	Arch	Known Vulnerabilities
Tenda	AC18	V15_03.05.19	ARM	CVE-2023-24164, CVE-2023-24165, CVE-2023-24166,CVE-2023-24167, CVE-2023-24169, CVE-2023-24170, CVE-2022-44171, CVE-2022-44172, CVE-2022-44174,CVE-2022-44175, CVE-2022-44176, CVE-2022-44177, CVE-2022-44178, CVE-2022-44180, CVE-2022-44183, CVE-2022-43260,CVE-2022-40861, CVE-2022-40854, CVE-2022-38309, CVE-2022-38310, CVE-2022-38311, CVE-2022-38312, CVE-2022-38313, CVE-2022-38314, CVE-2022-30472, CVE-2022-30473, CVE-2022-30474, CVE-2022-30475, CVE-2022-30476, CVE-2022-30477
	AC15	V1.0BR_V15.03.1.16	ARM	CVE-2018-5767
	AC8	V16.03.34.06	ARM	CVE-2024-11745, CVE-2024-10130, CVE-2024-10123, CVE-2024-46652, CVE-2024-4066, CVE-2024-4065, CVE-2024-4064, CVE-2023-4744, CVE-2023-40900, CVE-2023-40899, CVE-2023-40898, CVE-2023-40897, CVE-2023-40896, CVE-2023-40895, CVE-2023-40894, CVE-2023-40893, CVE-2023-40892, CVE-2023-40891, CVE-2023-39786, CVE-2023-39785, CVE-2023-39784, CVE-2023-38937, CVE-2023-3893, CVE-2023-38931, CVE-2023-33675, CVE-2023-33673, CVE-2023-33672, CVE-2023-33671, CVE-2023-33670, CVE-2023-33669
	AX1803	V1.0.0.1_2890	ARM	CVE-2022-40875, CVE-2022-40874, CVE-2022-40876, CVE-2022-37824, CVE-2022-37823, CVE-2022-37822, CVE-2022-37821, CVE-2022-37820, CVE-2022-37819, CVE-2022-37818, CVE-2022-37817, CVE-2022-30040
ASUS	RT-N53	V3_0.0.4.376.3754	ARM	CVE-2019-20082
	RT-N10LX	V2_0.0.39	MIPS	CVE-2023-34942 , CVE-2023-34940
D-link	DIR-600	B5_2.18	MIPS	CVE-2023-33626
	DIR-816	A2 v1.10CNB05	MIPS	CVE-2023-43242, CVE-2023-43240, CVE-2023-43239, CVE-2023-43238, CVE-2023-43237, CVE-2023-43236, CVE-2022-43003, CVE-2022-43002, CVE-2022-43001, CVE-2022-43000, CVE-2022-42998, CVE-2021-27114,CVE-2018-20305, CVE-2018-17067, CVE-2018-17065, CVE-2018-11013, CVE-2022-36620, CVE-2022-37134, CVE-2022-29327, CVE-2022-29326, CVE-2022-29325, CVE-2022-29324, CVE-2022-29323, CVE-2022-29322, CVE-2022-29321
	DAP-1360	V6.14b01	MIPS	CVE-2023-32136, CVE-2023-32138, CVE-2023-32140, CVE-2023-32141, CVE-2023-32142, CVE-2023-32144, CVE-2023-32146
H3C	GR-1200W	MiniGRW1A0V100R006	MIPS	CVE-2022-37074, CVE-2022-37073, CVE-2022-37072, CVE-2022-37071, CVE-2022-37069, CVE-2022-37068, CVE-2022-37067, CVE-2022-37066, CVE-2022-36520, CVE-2022-36519, CVE-2022-36518, CVE-2022-36517, CVE-2022-36516, CVE-2022-36515, CVE-2022-36514, CVE-2022-36513, CVE-2022-36511
Totolink	A860R	V4.1.2cu.5182	MIPS	CVE-2022-37842, CVE-2022-37839, CVE-2022-37840
Netgear	R6400	V2_1.0.4.118	ARM	CVE-2021-34991, CVE-2021-45525, CVE-2021-45527, CVE-2021-45549, CVE-2021-45554, CVE-2021-45604, CVE-2021-45606, CVE-2021-45607, CVE-2021-45610, CVE-2022-48196, CVE-2023-36187
TP-Link	TL-WR940N	V4	MIPS	CVE-2023-33536, CVE-2023-33537, CVE-2023-36354, CVE-2023-36355, CVE-2023-36356, CVE-2023-36357, CVE-2023-36358, CVE-2023-36359
Motorola	M2	V1.01	MIPS	CVE-2019-12297
Linksys	WRT54GL	V4.30.18.006	MIPS	CVE-2022-43970
Belkin	N750	F9K1103 v1.10.22	MIPS	CVE-2018-1145
TrenDnet	TEW-632BRP	V1.010B32	MIPS	CVE-2018-19242
Zyxel	VMG8825-T50K	V5.50(ABOM.8)C0	MIPS	CVE-2024-8748, CVE-2024-5412
Total	-	-	-	154

6.1. Effectiveness of Pre-analysis (RQ1)

We design this experiment to assess the effectiveness of DFirmSan's pre-analysis module in identifying and filtering sensitive detection points within the firmware.

The primary objective of this experiment is to assess DFirmSan's capability to extract all service programs and vendor-customized libraries from each firmware. We measure the time required for this analysis and evaluate its accuracy by analyzing false positives and false negatives in identifying these binaries. To further assess the pre-analysis module, we select a representative set of binary files from the firmware that are likely to contain vulnerabilities. This set includes executable files, CGI scripts, and libraries, as these file types frequently invoke sensitive functions. We then compare the number of sensitive detection points identified before and after applying untrusted data tracking. Additionally, we evaluate the accuracy of the sensitive function discrimination rules established in DFirmSan by analyzing false positives and false negatives in identifying sensitive function calls.

Experiment Result. Table 5 presents DFirmSan's extraction analysis re-

sults for all firmware. The analysis reveals that DFirmSan successfully extracts a total of 891 service programs and identifies 215 associated vendor-customized libraries. Overall, the size of a firmware package is positively correlated with the number of service programs that the firmware contains. However, there are exceptions. For example, the Belkin N750 firmware is only 7.4 MB but contains 85 service programs, which is potentially due to its unique packaging method. In contrast, the number of vendor-customized libraries shows limited correlation with firmware size, varying from just a few to several dozen. For instance, the Zyxel VMG8825-T50K firmware contains 47 vendor-defined libraries, indicating that it provides an extensive range of features. Focusing solely on memory monitoring for data input programs can result in the oversight of numerous vulnerabilities hidden within other programs and libraries. Typically, larger firmware packages offer more complex functionalities and contain a greater number of service programs and libraries. This complexity poses a challenge: if static rewriting is applied to all programs, it becomes difficult to maintain the integrity of the firmware's operational logic, leading to many libraries potentially being overlooked. To address this issue, DFirmSan employs dynamic detection methods while preserving the original state of the programs, aligning with a more effective approach to vulnerability detection.

We analyze the false positives and false negatives in identifying service programs and vendor-customized libraries within the firmware. For service program identification, we observe 29 false positives (3.25%) and 4 false negatives (0.45%). Similarly, for vendor-customized libraries, there are 6 false positives (2.79%) and 1 false negative (0.47%). Overall, the identification demonstrates high accuracy and coverage. While a small number of false positives are identified, their impact on performance overhead is negligible. These false positives are likely caused by incorrect recognition of function names in the binary, compiler optimizations, or insufficient contextual understanding, particularly in complex firmware like the Zyxel VMG8825-T50K. The few missed detections primarily result from encryption or code obfuscation techniques.

Table 5: Extract Results of Firmwares.

Vendor	Model	Firmware Version	Image Size	Arch	Service Programs			Vendor-customized Libraries			Analyse Time
					Num	FP	FN	Num	FP	FN	
Tenda	AC18	V15.03.05.19	10MB	ARM	35	2	0	14	0	0	09m11s
	AC15	V1.0BR_V15.03.1.16	10.5MB	ARM	24	0	0	12	0	0	07m59s
	AC8	V16.03.34.06	7.2MB	ARM	48	2	0	10	0	0	07m37s
ASUS	AX1803	V1.0.0.1_2890	38.4MB	ARM	98	3	0	17	1	0	19m26s
	RT-N53	V3_0.0.4.376.3754	7.02MB	ARM	24	0	0	18	0	0	07m31s
	RT-N10LX	v2_0.0.39	2.46MB	MIPS	17	0	2	3	0	0	05m45s
D-link	DIR-600	B5_2.18	3.6MB	MIPS	41	2	0	4	0	0	05m39s
	DIR-816	A2 v1.10CNB05	3.7MB	MIPS	38	0	1	6	0	0	06m18s
	DAP-1360	V6.14b01	3.7MB	MIPS	36	1	0	5	0	0	05m57s
H3C	GR-1200W	MiniGRW1A0V100R006	9.44MB	MIPS	31	0	0	11	0	0	06m30s
Totolink	A860R	V4_1.2cu.5182	4.74MB	MIPS	30	2	1	3	0	1	04m54s
Netgear	R6400	V2_1.0.4.118	45MB	ARM	73	3	0	15	0	0	13m27s
TP-Link	TL-WR940N	V4	4.1MB	MIPS	23	0	0	7	0	0	04m18s
Motorola	M2	V1.01	16.8MB	MIPS	66	1	0	18	0	0	11m14s
Linksys	WRT54GL	V4.30.18.006	3.5MB	MIPS	37	0	0	9	0	0	07m02s
Belkin	N750	F9K1103 v1.10.22	7.4MB	MIPS	85	2	0	10	0	0	10m28s
TrenDnet	TEW-632BRP	1.010B32	3.5MB	MIPS	31	0	0	6	0	0	06m29s
Zyxel	VMG8825-T50K	5.50(ABOM.8)C0	26.8MB	MIPS	154	11	0	47	5	0	23m46s
Total	-	-	-	-	891	29(3.25%)	4(0.45%)	215	6(2.79%)	1(0.47%)	2h43m31s

Moreover, DFirmSan completes a holistic pre-analysis of a single firmware package within 25 minutes, which is a relatively small fraction of the time re-

quired for the subsequent fuzzing process, typically lasting 24 hours or more. Notably, the analysis results can be reused, further enhancing efficiency. The total time spent analyzing all firmware packages in this experiment is less than 3 hours, indicating that the time overhead introduced by the pre-analysis phase is minimal, particularly in the context of large-scale automated fuzzing.

Table 6: Analysis Results of Binaries.

Firmware	Binary	Type	Size	Call of Sensitive Func			Untrusted Data Tracking		
				Total	FP	FN	Total	Type-I	Type-II
Tenda AC18	httpd	program	980KB	1007	0	1	348	296	52
Tenda AC15	httpd	program	784KB	781	0	0	271	238	33
	libcommon.so	library	74.1KB	51	1	0	24	17	7
Tenda AC8	httpd	program	1MB	954	2	4	407	381	26
Tenda AX1803	tdhttpd	program	2.28MB	491	1	2	246	246	0
ASUS RT-N53	httpd	program	226KB	206	1	0	121	112	9
	rc	program	551KB	501	1	3	250	249	1
ASUS RT-N10LX	httpd	program	471KB	667	1	2	245	234	11
D-Link DIR-600	cgibin	program	165KB	110	0	1	86	86	0
D-Link DIR-816	goahead	program	549KB	391	1	0	279	268	11
D-Link DAP-1360	webproc	program	90KB	121	0	0	63	61	2
H3C GR-1200W	webs	program	1.10MB	850	2	3	416	393	23
Totolink A830R	infostat.cgi	program	15.2KB	31	0	0	26	26	0
Netgear R6400	upnpd	program	515KB	763	2	1	564	554	10
	libupnp.so	library	35.6KB	13	0	0	13	13	0
TP-Link TL-WR940N	httpd	program	1.7MB	1587	5	0	816	798	18
Motorola M2	scopd	program	173.3KB	101	1	0	46	46	0
Linksys WRT54GL	httpd	program	622.6KB	1132	2	4	503	493	10
Belkin N750	libcfg.so	library	122.8KB	130	1	0	94	91	3
TrenDnet TEW-32BRP	httpd	program	321KB	522	0	2	366	362	4
Zyxel NBG-418N	libclinkc.so	library	417.4KB	378	0	0	49	49	0
Total	-	-	-	10787	21(0.19%)	23(0.21%)	5233(48.51%)	5013	220

Subsequently, Table 6 presents the number of sensitive function calls identified in the selected 21 binaries, highlighting the difference in counts before and after untrusted data tracking is implemented. Despite the generally small size of the firmware binaries, there is a significant prevalence of sensitive function misuse, with DFirmSan detecting a total of 10,787 calls of sensitive functions. For instance, the *httpd* service in the Tenda AC18, which is only 980KB in size, exhibits an alarming 1,007 calls to sensitive functions. In contrast to traditional desktop software, where the usage of such sensitive functions is typically restricted or closely monitored, note that risks associated with these functions are often overlooked in the development of IoT device firmware.

False Positives and False Negatives Analysis. We analyze the false positives and false negatives in identifying sensitive function calls to evaluate the effectiveness of the sensitive function recognition criteria we set. For false positives, we develop a script to display detailed information about all identified function call sites, including the function name, parameter types, and relevant contextual information. These call sites are manually audited to identify sensitive function calls that are incorrectly classified. As a result, out of all the calls, there are only 21 false positives (accounting for 0.19%). These can result from certain simple memory reads in compiler-optimized code being misidentified as complex memory-sensitive operations. For false negatives, we utilize IDA Pro to examine the binaries listed in Table 6 at the assembly level. Expert knowledge is applied to identify custom memory management functions in third-party libraries. We also search for code segments where reverse engineer-

ing tools perform poorly and inspect them to determine whether any sensitive function calls are missed. This analysis identified only 23 false negatives (accounting for 0.21%), which may occur because memory management functions in third-party libraries (such as custom allocators) are not detected, or because the disassembly engine fails to parse memory operations in inline assembly.

After applying the untrusted data tracking method, 5,233 detection points or 48.51% of the total number remain. These results demonstrate that, despite our over-untrusted strategy, we successfully eliminate most redundant detection points. This significant reduction helps reduce the overhead for subsequent dynamic detection. Among the final detection points, type-I sensitive functions represent 5,013 calls, indicating that most of the overhead incurred by DFirmSan during fuzzing is dedicated to monitoring these function calls. Although calls to type-II sensitive functions are less frequent, totaling only 220, their role in parsing incoming external data necessitates ongoing vigilance regarding their potential risks.

Answer to RQ1: DFirmSan successfully extracts 891 service programs and 215 libraries from 18 firmware. In less than 3 hours, it identifies 10,787 vulnerability detection points from 21 selected binaries. By employing untrusted data tracking, DFirmSan refines this to 5,233 critical detection points, effectively filtering out most of the redundant ones.

6.2. Efficiency of vulnerability discovery (RQ2)

This experiment evaluates the efficiency of DFirmSan in real-world vulnerability discovery scenarios. Specifically, we compare the number of vulnerabilities identified by different firmware fuzzers before and after integrating DFirmSan and other sanitizers, assessing their effectiveness and time overhead.

Table 4 details the CVE IDs corresponding to all the memory vulnerabilities we discover in the firmware dataset. It is important to note that device vendors may release firmware versions containing various patches or historical vulnerabilities at different times. Additionally, even firmware with the same version number may differ in code implementation based on the regions where the products are sold. The vulnerabilities presented in the table correspond solely to the firmware obtained through our data collection and analysis.

To ensure experimental rigor, we utilize two firmware fuzzers and compare DFirmSan with two dynamic sanitizers: Memcheck [5] and QASan [11]. The first fuzzer employed is AFL++ [46], combined with the multi-architecture emulation framework Qiling [43] to emulate Linux-based firmware. The second fuzzer is FirmAE [1], a sophisticated Linux-based firmware emulation framework with robust fuzzing capabilities. FirmAE uses an external probing approach to detect memory security vulnerabilities by analyzing feedback from the target web service. To integrate DFirmSan into this setup, we modify the source code of QEMU [25], the core emulation component of FirmAE. This allows us to evaluate whether DFirmSan enhances the vulnerability detection capabilities of the existing fuzzer. Because FirmAE requires a feedback waiting time of 10

Table 7: Vulnerability Detection Results for AFL++.

Vendor	Model	AFL++		AFL++ & DFirmSan		AFL++ & QASan		AFL++ & Memcheck	
		Alert	Speed (run/s)	Alert	Speed (run/s)	Alert	Speed (run/s)	Alert	Speed (run/s)
Tenda	AC18	4	13.77	22	11.43	7	7.45	2	1.29
	AC15	0	19.75	1	15.01	1	5.67	1	1.71
	AC8	2	15.45	23	11.95	5	6.46	1	1.19
ASUS	AX1803	2	420.55	12	399.72	7	95.36	4	37.87
	RT-N53	0	26.62	1	24.17	0	7.76	0	2.11
	RT-N10LX	1	28.41	2	23.81	1	7.48	1	1.95
D-link	DIR-600	0	1598.92	1	1482.34	1	773.81	1	212.46
	DIR-816	3	10.42	21	9.52	14	8.22	1	0.90
	DAP-1360	2	1714.40	8	1640.45	7	908.63	4	253.73
H3C	GR-1200W	2	481.7	16	466.35	11	114.65	2	41.87
Totolink	A830R	0	39.28	3	36.04	1	12.07	0	3.35
Netgear	R6400	1	21.11	10	20.55	6	15.75	1	1.17
TP-Link	TL-WR940N	2	263.48	8	221.31	5	110.76	3	35.57
Motorola	M2	1	57.65	1	44.39	1	19.48	1	4.78
Linksys	WRT54GL	0	19.56	1	15.77	1	6.34	0	1.01
Belkin	N750	0	933.75	1	392.45	1	128.18	1	79.47
TrenDnet	TEW-32BRP	0	70.15	1	51.21	1	27.36	0	3.90
Zyxel	VMG8825-T50K	0	13.38	2	10.17	0	2.54	0	0.98
Total	-	17	-	134(+117)	↓16.43%	70(+53)	↓61.55%	23(+6)	↓91.26%

Table 8: Vulnerability Detection Results for FirmAE.

Vendor	Model	FirmAE		FirmAE & DFirmSan		FirmAE & QASan		FirmAE & Memcheck	
		Alert	Avg-time (per-run)	Alert	Avg-time (per-run)	Alert	Avg-time (per-run)	Alert	Avg-time (per-run)
ASUS	RT-N53	1	12.93	1	13.04	0	14.13	0	15.98
	RT-N10LX	0	12.87	2	12.94	2	13.53	1	15.75
D-link	DIR-600	0	11.38	1	11.41	1	12.86	0	15.50
	DIR-816	3	13.97	9	14.05	5	14.29	2	15.31
	DAP-1360	0	11.43	5	11.97	2	12.71	1	14.39
Netgear	R6400	1	15.29	4	15.43	1	16.14	1	16.46
TP-Link	TL-WR940N	1	11.86	5	12.07	4	12.99	1	14.53
Linksys	WRT54GL	0	12.03	1	12.45	1	13.50	1	15.29
Belkin	N750	0	12.73	1	13.65	0	15.48	0	16.07
TrenDnet	TEW-32BRP	0	13.41	1	13.83	1	14.68	0	15.32
Zyxel	VMG8825-T50K	0	15.04	1	15.90	0	16.32	0	16.69
Total	-	6	12.99	31(+25)	13.34(+0.35)	17(+11)	14.24(+1.24)	7(+1)	15.57(+2.58)

to 15 seconds for a single fuzz test, we calculate its average test duration to gauge fuzzing speed. Each firmware sample is fuzzed for 24 hours across all experimental groups to ensure consistency in testing conditions. In addition, we ensure that all initial seeds are valid and have sufficient path exploration depth.

Experiment Result. Table 7 and Table 8 present the number of vulnerabilities identified during the fuzzing across all experimental groups, along with their respective testing efficiencies. All vulnerabilities in the tables are previously reported CVEs, to fairly evaluate the effectiveness of DFirmSan compared to other sanitizers. It is important to note that FirmAE exclusively tests web services within the firmware, whereas AFL++ evaluates all running services exposed in the Qiling emulation environment. This results in a notable disparity in the vulnerability discovery capabilities of the two fuzzers. Consequently, we focus our comparison solely on the results of each fuzzer with and without sanitizers, rather than comparing the two fuzzers with each other. In addition, some experiments are not demonstrated because the comparative FirmAE is unable

to emulate devices from four vendors: Tenda, H3C, Totolink, and Motorola.

We first analyze the number of vulnerabilities detected in each experimental group. Without any sanitizers, AFL++ identifies only 17 vulnerabilities, while FirmAE detects just 6 vulnerabilities. However, integrating DFirmSan leads to a significant improvement in detection capabilities for both fuzzers. Specifically, DFirmSan-empowered AFL++ identifies 134 vulnerabilities, an increase of 117 compared to the unmodified version. Similarly, DFirmSan-empowered FirmAE detects 31 vulnerabilities, an improvement of 25. The results clearly demonstrate DFirmSan's ability to enhance the effectiveness of firmware fuzzers in uncovering real-world vulnerabilities.

Next, we analyze the impact of DFirmSan on fuzzing speed. The results demonstrate that DFirmSan introduces only a minor performance overhead, keeping the reduction in fuzzing speed within an acceptable range. Specifically, for AFL++, the average fuzzing speed decreases by 16.43%. For FirmAE, the duration of a single test typically ranges from 11 to 15 seconds, primarily due to the time required to wait for feedback from the web service. After integrating DFirmSan, the testing time increases slightly, with an average rise of 0.35 seconds, equating to approximately a 2.69% overhead. These results confirm that DFirmSan maintains low time overhead while enhancing vulnerability detection.

Overall, the time overhead after using DFirmSan is $1-3\times$ that of the original program. This aligns with theoretical estimates. However, this increase is minimal in practical terms. This is largely due to the numerous time-consuming file I/O and peripheral read/write operations inherent in firmware runtime, whereas most of DFirmSan's processes occur at the memory level, with only a few additional reads and writes to the pre-analysis report, with negligible extra time overhead.

Compare with QASan and Memcheck. When integrated with AFL++, DFirmSan demonstrates significantly better performance overhead than that of other sanitizers. Specifically, QASan slows down testing speed by 61.55%, and Memcheck by 91.26%, whereas DFirmSan incurs only a 16.43% slowdown. This better overhead enables DFirmSan to uncover far more vulnerabilities. While QASan identifies 53 additional vulnerabilities and Memcheck only 6, DFirmSan detects 117, showcasing its superior effectiveness. This issue is even more pronounced with FirmAE. QASan and Memcheck increase the duration of a single test by 1.24 and 2.58 seconds, respectively, compared to DFirmSan's modest 0.35-second increase. QASan and Memcheck uncover only 11 and 1 additional vulnerabilities, respectively, whereas DFirmSan identifies 25. In particular, due to the significant slowdown in test speed, Memcheck's test results exhibit a marginal enhancement in comparison to the original fuzzers. These results highlight the critical importance of balancing memory corruption detection with runtime performance. DFirmSan's lightweight design ensures minimal performance impact while maintaining superior vulnerability detection.

Answer to RQ2: DFirmSan significantly enhances the vulnerability discovery capabilities of fuzzers, enabling AFL++ and FirmAE to identify 117 and 25 additional known CVEs, respectively. The time overhead remains low, at 16.43% for AFL++ and 2.69% for FirmAE.

6.3. Capability of Crash Detection (RQ3)

This section evaluates DFirmSan's ability to detect memory corruption caused by real-world CVE vulnerabilities, validating its effectiveness in enhancing memory detection for firmware fuzzers. Additionally, we compare DFirmSan with QASan and Memcheck to demonstrate that it achieves detection capabilities on par with existing state-of-the-art sanitizers.

We further analyze the fuzzing results from Section 6.2 by selecting one or more CVE memory vulnerabilities from each firmware sample. This allows us to create a representative sample set that encompasses a variety of memory issues, including buffer overflow (CWE-119), stack overflow (CWE-121), heap overflow (CWE-122), format string vulnerabilities (CWE-134), and out-of-bounds writes (CWE-787). For each identified vulnerability, we extract the associated Proof-of-Concepts (PoCs) and manually review any reported false positives. To evaluate DFirmSan's ability in detecting Silent Memory Corruptions (Silent-MCs), we input all PoCs into the emulated firmware and count the Silent-MCs that do not lead to any observable firmware alerts.

Result Analysis. Table 9 summarizes the details of each CVE vulnerability, including the corresponding types of sensitive functions and the outcomes of the fuzzing experiments. Without sanitizers, the fuzzer can only detect five trivially triggered stack overflow vulnerabilities, producing a total of 62 Proof-of-Concepts (PoCs). In contrast, integrating DFirmSan significantly improves detection capabilities, allowing the fuzzer to identify all 21 vulnerabilities and generate 3,408 PoCs—an improvement of over 54×. Notably, our analysis reveals that 3,338 of these PoCs (over 97%) trigger Silent-MCs, which are difficult to observe in standard firmware behavior. This highlights that even with a powerful fuzzing engine capable of triggering vulnerabilities, the absence of an effective detection mechanism can hinder vulnerability identification. For example, CVE-2022-40875 involves sensitive variables stored in dynamic heap space, making it particularly challenging to detect during a heap buffer overflow. By introducing DFirmSan, we successfully identify this vulnerability and generate 214 PoCs, all of which trigger Silent-MCs, further affirming DFirmSan's effectiveness in uncovering hidden vulnerabilities.

Notably, DFirmSan successfully identifies the stealthy vulnerability CVE-2019-20082, which involves multiple service programs within the firmware, as illustrated in Figure 5. In this case, externally input data is first stored in the configuration file NVRAM by the program *httpd*, subsequently read from NVRAM by the program *rc*, and ultimately leads to a *buffer overflow* through sensitive functions such as *strcpy* or *sprintf*. Generating a test case that can effectively crash the program through random mutations is challenging due to

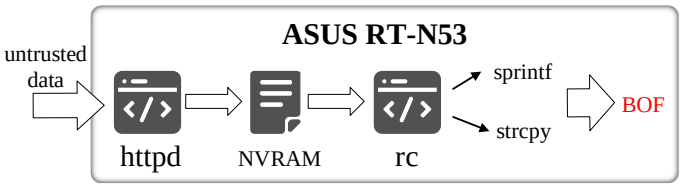

Figure 5: Analysis of CVE-2019-20082.

the specific propagation path of the untrusted data. However, with the memory detection mechanism provided by DFirmSan, even if the generated seeds only cause minor memory corruption, they can still be recognized. This significantly reduces the requirement for fuzzer mutation capabilities, thereby facilitating the detection of such complex vulnerabilities.

Table 9: PoCs Analysis for CVEs.

ID	Info	MemLoc	Sens-Func	Type	Original	With DFirmSan			With QASan			With Memcheck		
					PoCs	PoCs	FP	Silent	PoCs	FP	Silent	PoCs	FP	Silent
CVE-2023-24164		Stack	strcpy	I	0	152	0	152	66	3	25	32	8	24
CVE-2022-38310	Tenda AC18	Stack	sscanf	II	0	377	1	362	205	0	205	188	12	176
CVE-2022-38314		Heap	strcpy	I	0	91	1	90	87	0	87	29	0	29
CVE-2018-5767	Tenda AC15	Stack	sscanf	II	0	217	0	217	114	1	99	23	0	19
CVE-2024-11745	Tenda AC8	Stack	sscanf	II	0	97	0	95	41	0	41	35	0	30
CVE-2022-40875	Tenda AX1803	Heap	strcpy	I	0	214	0	214	53	2	51	34	0	34
CVE-2022-40876		Stack	strcpy	I	12	376	1	357	83	2	81	90	4	86
CVE-2019-20082	ASUS RT-N53	Stack	strcpy/sprintf	I	0	13	0	13	0	-	-	0	-	-
CVE-2023-34940	ASUS RT-N10LX	Stack	strcpy	I	4	226	0	221	135	2	133	43	5	38
CVE-2023-33626	D-Link DIR-600	Stack	sprintf	I	0	95	0	91	60	0	47	55	2	53
CVE-2023-43238	D-Link DIR-816	Stack	strcat	I	0	61	0	61	27	0	27	16	1	15
CVE-2023-32136	D-Link DAP-1360	Stack	strcpy	I	26	218	0	214	258	2	205	171	11	160
CVE-2022-37074	H3C GR-1200W	Stack	sscanf	II	0	168	3	162	106	0	88	31	0	26
CVE-2022-37842	Totolink A830R	Stack	fread	I	0	34	1	33	11	0	11	0	-	-
CVE-2021-34991	Netgear R6400	Stack	strncpy	I	0	71	2	69	56	2	54	27	3	24
CVE-2023-36355	TP-Link TL-WR940N	Stack	strcpy	I	11	309	1	304	127	0	115	90	0	90
CVE-2019-12297	Motorola M2	Stack	snprintf	I	9	102	1	101	85	0	85	48	0	48
CVE-2022-43970	Linksys WRT54GL	Stack	sprintf	I	0	263	0	263	121	0	97	26	0	26
CVE-2018-1145	Belkin N750	Stack	sprintf	I	0	173	1	168	142	1	141	76	4	72
CVE-2018-19242	TrenDnet TEW-32BRP	Stack	sprintf	I	0	132	0	132	87	0	61	0	-	-
CVE-2024-8748	Zyxel VMG8825-T50K	Heap	sscanf	II	0	19	0	19	0	-	-	0	-	-
Total	-	-	-	-	62	3408	12(0.35%)	3338	1864	15(0.80%)	1653	1014	50(4.93%)	950

False Positive Analysis. False positive is a critical metric for evaluating the effectiveness of sanitizers, prompting us to implement Dynamic Boundary Update in DFirmSan to mitigate their occurrence. The results indicate that among identified 3,408 PoCs, only 12 cause false positives, leading to an overall false positive rate of less than 0.35%. These false positive PoCs stem from CVE exploit points or other detection points encountered along the execution path. Since these candidate false positives are clearly flagged, the time and effort needed to troubleshoot the experimental results are minimal. We conduct a further analysis of the causes behind these false positives, which confirms our theoretical expectations. In total, we categorize these false positives into two distinct types:

Case 1: Stack variable boundary misreporting. It can arise from several factors. For instance, limitations in the reverse engineering process may hinder the analysis of specific firmware due to the unique architecture or compiler settings of the device, particularly in the firmware of H3C routers. Additionally, when analyzing firmware with a large number of detection points, incorrect

inference of the memory size of variables may occur, as seen with the Netgear R6400. These inaccuracies increase the likelihood of false positives. DFirmSan addresses the issue by implementing dynamic boundary expansion, subsequently recording the discrepancy in the *check bytes* data at the updated detection point as a candidate false positive (not real FP). This approach ensures that the sensitive parameters at detection points that generate false positives are not impacted by boundary errors; instead, they are safeguarded by the expanded boundaries, effectively limiting each location to a single false positive.

Case 2: False positives can also arise when data discrepancies occur in parameters located in contiguous memory spaces. This issue is evident in the test results for vulnerability CVE-2022-38310, where multiple sensitive parameter variables (*s1*, *v9*, *v10*, *v11*) are modified at the detection point during execution (as shown in Listing 2). Due to the close proximity of these parameters in memory, the detected discrepancies in the *check bytes* data for variables v9, v10, and v11 are normal variations rather than indicators of memory corruption. Consequently, the input for this execution is recorded as a candidate FP. Following the dynamic merging and expansion of boundaries, the same detection point is maintained, but the new boundary is no longer influenced by memory space proximity or the issues identified in Case 1. As a result, any subsequent discrepancies detected in the *check bytes* data are not classified as false positives.

Compare with QASan and Memcheck. DFirmSan demonstrates comparable performance to QASan and Memcheck in generating PoCs, producing a total of 3,408 PoCs. QASan and Memcheck generate 1,864 and 1,014 PoCs, respectively, with silent PoCs accounting for 88.68% and 93.69% of their totals. DFirmSan successfully detects all 21 vulnerabilities, while QASan and Memcheck fail to identify vulnerabilities in several firmware, resulting in zero PoCs for those cases. Despite runtime speed constraints limiting the code exploration efficiency of fuzzers, QASan and Memcheck effectively detect vulnerabilities within their exploration range. These results affirm that DFirmSan achieves detection capabilities comparable to those of existing sanitizers. In terms of false positive rates, DFirmSan outperforms both QASan and Memcheck, with a significantly lower rate of 0.35% than that of QASan's 0.80% and Memcheck's 4.93%. The higher false positive rates of QASan and Memcheck can be attributed to factors such as incorrect pointer alias analysis, unusual memory access patterns, limited support for specific language features, and configuration issues. This highlights DFirmSan's improved precision in detecting vulnerabilities.

Answer to RQ3: DFirmSan significantly enhances the detection capabilities for memory corruption. During fuzzing of 21 CVE vulnerabilities, the number of effective PoCs increases from 62 to 3,408. Notably, only 12 false positives (0.35%) are recorded, while over 97% of the PoCs trigger silent memory corruptions that are undetectable by default mechanisms.

6.4. Scalability (RQ4)

This section evaluates the scalability of DFirmSan, demonstrating through experiments whether it is suitable for various firmware environments and capable of uncovering zero-day vulnerabilities in Linux-based firmware.

To evaluate the scalability of DFirmSan across various firmware complexities and sizes, we conduct large-scale tests using a publicly available dataset [47] (NDSS'24) on taint analysis for Linux-based IoT firmware vulnerability detection. This dataset includes 47 firmware samples from 20 product series across 8 major IoT vendors, as detailed in Table 10. We run parallel testing on multiple hosts, executing multiple processes with 12-hour fuzzing sessions for each firmware. The collected data is then analyzed to compare the number of crashes detected by the original fuzzer with those identified after integrating DFirmSan, demonstrating its scalability. Since the research corresponding to this dataset employs a static analysis method that does not produce crashes, we do not perform a comparison with it.

To further demonstrate DFirmSan's effectiveness, we apply DFirmSan to uncover zero-day vulnerabilities in real-world firmware. The fuzzer integrated with DFirmSan was used to analyze additional firmware samples that previous efforts had not thoroughly examined.

Table 10: Test Results on Large-scale Dataset.

Vendor	Device Series	Samples	Detection Points	Crash PoCs	
				Orginal	DFirmSan
ASUS	GT/RT	4	2.28×10^5	35	243
D-Link	COVR/DIR	3	3.48×10^4	0	395
Linksys	E/EA/MR	3	4.40×10^5	0	0
Netgear	MR/RAX	4	4.85×10^5	0	0
Totolink	A/LR/T/X	8	4.14×10^4	175	2608
TP-Link	AX/XDR/XTR	6	3.57×10^5	0	0
Tenda	AC/AX/W	10	3.27×10^5	358	8170
TrenDnet	TEW	9	3.18×10^5	217	4792
Total	/	47	1.82×10^6	785	16208
Average	/	/	3.87×10^4	16.70	344.85

Experiment Result. Table 10 summarizes the results of large-scale testing. Across 47 firmware samples, DFirmSan identifies and filters 1.82×10^6 sensitive function calls, using them as detection points for memory vulnerabilities. After extensive testing, the fuzzer integrated with DFirmSan detects 26,208 PoCs triggering memory crashes—an increase of 25,423 compared to the fuzzer alone. On average, DFirmSan-enhanced fuzzing uncovers 344.85 crashes per firmware sample, more than 20 times the number detected by the original fuzzer. These results highlight DFirmSan's ability to scale effectively across diverse firmware datasets.

After verifying the scalability of DFirmSan, we apply it to further firmware

vulnerability mining and successfully uncover several zero-day vulnerabilities. Exploits are then developed for these vulnerabilities and submitted to a professional vulnerability platform for validation. Consequently, they are assigned 4 CNVD IDs [3], specifically: For the D-Link DAP-1360 device, we uncover a high-risk buffer overflow vulnerability, registered as CNVD-2024-35297. For the Tenda AC8, we uncover a memory corruption vulnerability that could lead to a potential DoS attack, assigned CNVD-2024-36681. For the H3C Magic B5 and N12 devices, we find two stack overflow vulnerabilities, assigned CNVD-2023-15074 and CNVD-2023-16187, respectively. Notably, the original fuzzers, as well as those integrated with QASan or Memcheck, are unable to detect these vulnerabilities, even with sufficient test time. These findings demonstrate DFirmSan's robust memory detection capabilities and significant contribution to identifying critical vulnerabilities in real-world IoT devices.

Answer to RQ4: In the large-scale dataset, the fuzzer integrated with DFirmSan detects 16,208 crashes, identifying over 20 times more crashes per firmware sample on average. Additionally, DFirmSan helps uncover four CNVD vulnerabilities. These results highlight its scalability and effectiveness in real-world vulnerability detection.

7. Discussion & Limitation

Untrusted Data Entrance. Identifying untrusted data entrances in Linux-based firmware during the *pre-analysis of sensitive info* is a challenging task. The design of these data entrances can vary due to vendor specifications and developer coding practices, often resulting in different network data receiving functions or naming conventions for global variables. Failing to identify these data entrances may lead to overlooking potentially vulnerable detection points. Fortunately, similar data entrances are frequently used across products from the same vendor or within the same product series. Thus, we established a comprehensive set of predefined untrusted data entrances based on domain knowledge, enhancing the applicability of DFirmSan to a wide range of Linux-based firmware. Furthermore, we included an option for *untrusted data tracking*, allowing users to customize their untrusted data entrances to reduce the likelihood of false negative.

Other Types of Vulnerabilities. DFirmSan was designed to detect memory vulnerabilities while minimizing memory and time overhead. However, this does not restrict its applicability to only memory-related vulnerabilities. The sensitive function-based approach employed in DFirmSan can be extended to various types of vulnerabilities. For example, to address command injection vulnerabilities often seen in IoT devices, we can expand the types of sensitive

[3] For security reasons, the specific details of these vulnerabilities are not disclosed at this time, pending the official release of the remediation plan and patches.

functions to adjust the detection points. This allows us to identify potential command injection by monitoring sensitive parameters that are influenced by fuzz inputs during runtime. Future research can build on this methodology to explore additional vulnerability types. And to minimize additional overhead in fuzz testing, we recommend using multiple fuzzing processes to identify different types of vulnerabilities rather than integrating all detection mechanisms at once.

Improving Fuzzer with DFirmSan. Various studies have utilized the functions of sanitizers to enhance the effectiveness and efficiency of fuzz testing. For instance, some fuzzers utilize AddressSanitizer to gain insights into the running state of programs, thereby increasing coverage and deepening program path exploration [48, 49]. Consequently, leveraging DFirmSan to enhance the vulnerability detection of firmware fuzzers presents a promising avenue for future research. We can optimize the energy allocation process in fuzzing by fully utilizing the characteristics of sensitive functions. This includes assigning different weights to detection points and using dynamic feedback from these points to calculate seed value scores.

Limitation of Binary Analysis Tools. The binary analysis tools employed in our approach, including IDA Pro and objdump, are essential for disassembling binaries but come with limitations. These tools may incorrectly interpret certain instructions, particularly in complex architectures, and struggle with machine code altered by compiler optimizations, making it challenging to map back to the original source code. Additionally, their limited support for specific file formats can result in incomplete or inaccurate disassembly for certain binary variants or implementations. The development of more robust tools in the future would greatly benefit this field.

Threat to Validation. Several factors could affect the validity of our approach. First, DFirmSan currently supports only firmware binaries developed in C and C++. Second, the accuracy of the disassembly engine can affect the results of data flow analysis for complex programs. Third, as crashes are manually verified, there is a risk that some vulnerabilities may remain undetected.

8. Conclusion

This work introduces DFirmSan, an innovative dynamic sanitizer designed to identify memory vulnerabilities in Linux-based firmware. DFirmSan employs a *dynamic boundary checking* method to conduct fine-grained memory monitoring of service programs and vendor-customized libraries, enabling the detection of unobservable *silent memory corruption*. To minimize additional overhead and ensure the firmware operates correctly, DFirmSan conducts a holistic *pre-analysis of sensitive function*, focusing memory monitoring on parameters most likely affected by untrusted data. This targeted approach enhances efficiency without compromising functionality. Moreover, to address the limitations of boundary checks and incorrect boundaries associated with sensitive parameters, we have implemented a *dynamic boundary update* method to mitigate the risk of

false positives. Evaluation results demonstrate that DFirmSan significantly enhances the vulnerability discovery capabilities of various firmware fuzzers while maintaining low overhead and a minimal false positive rate.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (62372236, 62402223), the Open Foundation of the State Key Laboratory of Integrated Services Networks (ISN24-15), the Postdoctoral Fellowship Program of CPSF (GZB20240982), the Qing Lan Project of Jiangsu Province, and the Jiangsu Funding Program for Excellent Postdoctoral Talent.

References

[1] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, Y. Kim, Firmae: Towards large-scale emulation of iot firmware for dynamic analysis, in: Annual computer security applications conference, 2020, pp. 733–745.

[2] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, A. Abbasi, Fuzzware: Using precise mmio modeling for effective firmware fuzzing, in: 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 1239–1256.

[3] T. Scharnowski, S. Wörner, F. Buchmann, N. Bars, M. Schloegel, T. Holz, Hoedur: embedded firmware fuzzing using multi-stream inputs, in: 32nd USENIX Conference on Security Symposium (USENIX Security 23), 2023, pp. 2885–2902.

[4] X. Chen, Y. Shi, Z. Jiang, Y. Li, R. Wang, H. Duan, H. Wang, C. Zhang, Mtsan: A feasible and practical memory sanitizer for fuzzing cots binaries, in: 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 841–858.

[5] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, ACM Sigplan notices 42 (6) (2007) 89–100.

[6] K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov, Addresssanitizer: A fast address sanity checker, in: 2012 USENIX annual technical conference (USENIX ATC 12), 2012, pp. 309–318.

[7] S. Dinesh, N. Burow, D. Xu, M. Payer, Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization, in: 2020 IEEE Symposium on Security and Privacy (SP), IEEE, 2020, pp. 1497–1511.

[8] M. Salehi, D. Hughes, B. Crispo, μsbs: Static binary sanitization of bare-metal embedded devices for fault observability, in: 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020), 2020, pp. 381–395.

[9] J. Shi, W. Li, W. Wang, L. Guan, Facilitating non-intrusive in-vivo firmware testing with stateless instrumentation, in: Network and Distributed System Security Symposium (NDSS), 2024.

[10] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, M. Franz, Sok: Sanitizing for security, in: 2019 IEEE Symposium on Security and Privacy (SP), IEEE, 2019, pp. 1275–1295.

[11] A. Fioraldi, D. C. D'Elia, L. Querzoni, Fuzzing binaries for memory safety errors with qasan, in: 2020 IEEE Secure Development (SecDev), IEEE, 2020, pp. 23–30.

[12] S. Ramachandran, J. Rami, A. Shah, K. Kim, D. M. Rathod, Defence against crypto-ransomware families using dynamic binary instrumentation and dll injection, International Journal of Electronic Security and Digital Forensics 15 (4) (2023) 424–442.

[13] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, M. Payer, Firmfuzz: Automated iot firmware introspection and analysis, in: Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, 2019, pp. 15–21.

[14] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, L. Sun, Firm-afl:high-throughput greybox fuzzing of iot firmware via augmented process emulation, in: 28th USENIX Security Symposium (USENIX Security 19), 2019, pp. 1099–1114.

[15] E. Stepanov, K. Serebryany, Memorysanitizer: fast detector of uninitialized memory use in c++, in: 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, 2015, pp. 46–55.

[16] K. Serebryany, T. Iskhodzhanov, Threadsanitizer: data race detection in practice, in: Proceedings of the workshop on binary instrumentation and applications, 2009, pp. 62–71.

[17] M. Muench, J. Stijohann, F. Kargl, A. Francillon, D. Balzarotti, What you corrupt is not what you crash: Challenges in fuzzing embedded devices, in: Network and Distributed System Security Symposium (NDSS), 2018.

[18] X. Feng, X. Zhu, Q.-L. Han, W. Zhou, S. Wen, Y. Xiang, Detecting vulnerability on iot device firmware: A survey, IEEE/CAA Journal of Automatica Sinica 10 (1) (2022) 25–41.

[19] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu, et al., One bad apple spoils the barrel: Understanding the security risks introduced by third-party components in iot firmware, IEEE Transactions on Dependable and Secure Computing 21 (3) (2023) 1372–1389.

[20] L. Cavallaro, R. Sekar, Taint-enhanced anomaly detection, in: International Conference on Information Systems Security, Springer, 2011, pp. 160–174.

[21] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, Z. Liang, Dtaint: detecting the taint-style vulnerability in embedded device firmware, in: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, 2018, pp. 430–441.

[22] Z. Gui, H. Shu, F. Kang, X. Xiong, Firmcorn: Vulnerability-oriented fuzzing of iot firmware via optimized virtual execution, IEEE Access 8 (2020) 29826–29841.

[23] L. Chen, Y. Wang, Q. Cai, Y. Zhan, H. Hu, J. Linghu, Q. Hou, C. Zhang, H. Duan, Z. Xue, Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems, in: 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 303–319.

[24] L. D. Bartolomeo, H. Moghaddas, M. Payer, ARMore: Pushing love back into binaries, in: 32nd USENIX Security Symposium (USENIX Security 23), USENIX Association, Anaheim, CA, 2023, pp. 6311–6328.

[25] F. Bellard, Qemu, a fast and portable dynamic translator., in: USENIX annual technical conference, FREENIX Track, Vol. 41, California, USA, 2005, pp. 10–5555.

[26] B. Feng, A. Mera, L. Lu, P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling, in: 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 1237–1254.

[27] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, M. Payer, Halucinator: Firmware rehosting through abstraction layer emulation, in: 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 1201–1218.

[28] W. Zhou, L. Guan, P. Liu, Y. Zhang, Automatic firmware emulation through invalidity-guided knowledge inference, in: 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 2007–2024.

[29] L. Seidel, D. C. Maier, M. Muench, Forming faster firmware fuzzers., in: USENIX Security Symposium (USENIX Security 23), 2023, pp. 2903–2920.

[30] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, K. Zhang, Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing, in: Network and Distributed System Security Symposium (NDSS), 2018.

[31] N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, G. Vigna, Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices, in: 2021 IEEE Symposium on Security and Privacy (SP), IEEE, 2021, pp. 484–500.

[32] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, Y. Xiang, Snipuzz: Black-box fuzzing of iot firmware via message snippet inference, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 337–350.

[33] D. D. Chen, M. Woo, D. Brumley, M. Egele, Towards automated dynamic analysis for linux-based embedded firmware, in: Network and Distributed System Security Symposium (NDSS), 2016.

[34] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, L. Sun, Efficient greybox fuzzing of applications in linux-based iot devices via enhanced user-mode emulation, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, pp. 417–428.

[35] G. Hernandez, M. Muench, D. Maier, A. Milburn, S. Park, T. Scharnowski, T. Tucker, P. Traynor, K. R. B. Butler, Firmwire: Transparent dynamic analysis for cellular baseband firmware, in: Symposium on Network and Distributed System Security (NDSS), 2022.

[36] Y.-T. Cheng, S.-M. Cheng, Firmulti fuzzer: Discovering multi-process vulnerabilities in iot devices with full system emulation and vmi, in: Proceedings of the 5th Workshop on CPS&IoT Security and Privacy, 2023, pp. 1–9.

[37] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, G. Vigna, Karonte: Detecting insecure multi-binary interactions in embedded firmware, in: 2020 IEEE Symposium on Security and Privacy (SP), IEEE, 2020, pp. 1544–1561.

[38] K. Cheng, Y. Zheng, T. Liu, L. Guan, P. Liu, H. Li, H. Zhu, K. Ye, L. Sun, Detecting vulnerabilities in linux-based embedded firmware with sse-based on-demand alias analysis, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 360–372.

[39] S. H. Kim, C. Sun, D. Zeng, G. Tan, Refining indirect call targets at the binary level., in: Symposium on Network and Distributed System Security (NDSS), 2021.

[40] Hex-Ray, A powerful disassembler and a versatile debugger, `Available: https://hex-rays.com/ida-pro/` (Apr.4,2022).

[41] Angr, Angr, `Available:http://angr.io/` (2018).

[42] H. J. Tay, K. Zeng, J. M. Vadayath, A. S. Raj, A. Dutcher, T. Reddy, W. Gibbs, Z. L. Basque, F. Dong, Z. Smith, et al., Greenhouse: Single-service rehosting of linux-based firmware binaries in user-space emulation, in: 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 5791–5808.

[43] Q. F. project, Qiling, https://docs.qiling.io/en/latest/ (2019).

[44] Mordor Intelligence, Wireless router market size & share analysis - growth trends & forecasts (2025 - 2030), accessed: 2025-01-05 (2025).
URL https://www.mordorintelligence.com/industry-reports/wireless-router-market

[45] W. Fei, H. Ohno, S. Sampalli, A systematic review of iot security: Research potential, challenges, and future directions, ACM Computing Surveys 56 (5) (2023) 1–40.

[46] A. Fioraldi, D. Maier, H. Eißfeldt, M. Heuse, Afl++: Combining incremental steps of fuzzing research, in: 14th USENIX Workshop on Offensive Technologies (WOOT 20), 2020.

[47] Z. Gao, C. Zhang, H. Liu, W. Sun, Z. Tang, L. Jiang, J. Chen, Y. Xie, Faster and better: Detecting vulnerabilities in linux-based iot firmware with optimized reaching definition analysis, in: Proceedings of the 2024 Network and Distributed System Security Symposium, San Diego, CA, USA, Vol. 26, 2024.

[48] R. K. Medicherla, M. Nagalakshmi, T. Sharma, R. Komondoor, Hdrfuzz: Detecting buffer overruns using addresssanitizer instrumentation and fuzzing, arXiv preprint arXiv:2104.10466 (2021).

[49] Y. Li, S. Ji, C. Lyu, Y. Chen, J. Chen, Q. Gu, C. Wu, R. Beyah, V-fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs, IEEE Transactions on Cybernetics 52 (5) (2020) 3745–3756.