Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

AflIot: Fuzzing on linux-based IoT device with binary-level instrumentation

Xuechao Du[a], Andong Chen[a], Boyuan He[b], Hao Chen[c,1], Fan Zhang[a,d,1,*], Yan Chen[b,2]

[a] College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China
[b] Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208, USA
[c] Department of Computer Science, University of California, Davis, CA 95616, USA
[d] Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province, Hangzhou, China, 310027

ARTICLE INFO

Article history:
Received 1 September 2020
Revised 8 June 2022
Accepted 18 August 2022
Available online 23 August 2022

Keywords:
Greybox fuzzing
Internet of things
On-device
Binary-level
Daemon analysis

ABSTRACT

In recent years, *coverage-guided greybox fuzzing* has demonstrated its efficiency in detecting security vulnerabilities on traditional devices. Instrumentation information plays a significant role in sophisticated greybox fuzzer such as American Fuzzing Lop to directionally improve coverage and distill seeds. While open-source programs leverage wrapped assemblers to glean instrumentation information, closed-source programs can utilize the emulation-based instrumentation for coverage-guided fuzzing. The pervasiveness of the closed source puts a strong demand for emulation instrumentation. However, the required access to peripherals brings great difficulty in fuzzing on the emulator, especially for those various IoT devices. This paper presents AflIot, the first generic on-device fuzzing framework for Linux-based IoT binary programs. By leveraging offset-free binary-level instrumentation, binary programs can avoid unnecessarily rewriting, inherit compatibility of peripherals, and be executed directly on IoT devices by AflIot. We evaluate AflIot on multiple benchmarks with real-world IoT programs. AflIot identified 437 unique crashes in 13 binary programs, including 95 newly confirmed unique crashes. Those crashes demonstrate that AflIot is efficient and effective in detecting potential software bugs in binary programs on Linux-based IoT devices.

1. Introduction

Although it has been decades since the concept of the Internet of Things (IoT) was first presented, only in recent years have we witnessed an outbreak of its application in our daily lives. Various types of Commercial-Off-The-Shelf (COTS) IoT devices emerged rapidly, such as home routers, IP cameras, printers, smart TVs, lamps, fridges, and air conditioners. These devices extend Internet connectivity beyond traditional ones (e.g., computers and smartphones) to any object we may use. Thus IoT dramatically expands the capability of the Internet and provides convenience for everyone.

However, the widespread application of IoT devices also brings new challenges to security and privacy. The interconnection of these devices that are initially isolated expands the attacking surface significantly and makes IoT devices more vulnerable to cyber-attacks. For example, privilege escalation vulnerabilities have recently been found in the smart lock and can be used by attackers to break the authentication (loc, 0000). Even worse, Nie et al. (2017) have demonstrated that it is possible to hack into a car and take over the remote control of Engine Control Units (ECU) through a phishing Wi-Fi hotspot.

Many security assessment works have been devoted to finding software vulnerabilities and flaws in COTS devices to mitigate those security problems. Costin et al. (2014) leveraged static analysis techniques to acquire and analyze firmware automatically. They used specific patterns to search for existing security flaws. Although their approach is efficient and scalable, the static analyses they applied are too simple and do not involve any code analysis technique, which is insufficient to find software vulnerabilities and flaws. On the other side, Zaddach et al. (2014) took a dynamic fuzzing approach on embedded devices through serial GDB protocol (Alves, 0000). They monitored much of the firmware execution on a hybrid system combining an emulator with real hardware.

Besides, a generic dynamic fuzzing framework, American Fuzzy Lop (AFL) (Zalewski, 0000), also provides support for whitebox

* Corresponding author.
E-mail addresses: xcdu@zju.edu.cn (X. Du), chenandong@zju.edu.cn (A. Chen), bhe@northwestern.edu (B. He), chen@ucdavis.edu (H. Chen), fanzhang@zju.edu.cn (F. Zhang), ychen@northwestern.edu (Y. Chen).
[1] Member, IEEE
[2] Fellow, IEEE

https://doi.org/10.1016/j.cose.2022.102889
0167-4048/© 2022 Elsevier Ltd. All rights reserved.

and greybox fuzzing. Specifically, it constructs greybox fuzzing on top of Quick Emulator (QEMU) binary translation (Liaw, 0000) and records instrumentation information by the native support from QEMU. Well-known greybox fuzzers include AFL itself (Zalewski, 0000) and other AFL-based ones such as AFLFast (Böhme et al., 2017b), AFLGo (Böhme et al., 2017a), and FairFuzz (Lemieux and Sen, 2018), all of which share the advantages and disadvantages of the QEMU emulator. On one side, for those binary programs that can execute in the emulator, those fuzzers can apply their dynamic security assessment capability by leveraging the instrumentation information returned from emulated devices. However, on the other side, those greybox fuzzers will naturally inherit characteristics from QEMU and thus suffer its intrinsic incapability of emulating unsupported devices. Furthermore, there are other replaced[id=xcdu]works workds such as Firm-AFL (Zheng et al., 2019), PeriScope (Song et al., 2019), and Pretender (Gustafson et al., 2019) trying to leverage special mechanisms, such as MMIO and DMA, to detect communications from the peripherals or communicate with the virtual host.

Fuzzing has already been studied for years (Bastani et al., 2017; Böhme et al., 2017b; Cadar et al., 2008; Cha et al., 2015; Ganesh et al., 2009; Godefroid et al., 2005; 2012; 2017; Haller et al., 2013; Rawat et al., 2017; Sen et al., 2005; Stephens et al., 2016; Wang et al., 2017; 2010; Zheng et al., 2019), (Dinesh et al., 2020; Song et al., 2019). However, we still rarely identify its application in IoT devices nowadays. To help bridge this gap, we take a different approach instead, which applies the fuzzing techniques directly onto IoT devices.

Compared with its application on x86 platforms, fuzzing on IoT devices is facing several significant challenges:

1) **Highly diverse and limited hardware and software.** Unlike most x86 platform devices, IoT devices have a greater diversity in hardware and software. Even for Linux-based IoT devices, the hardware interfaces and system configurations vary among devices significantly. However, it is entirely in demand to have a generic fuzzing solution for IoT devices, which has to manage such diversities. Besides, compared to the x86 platform, IoT devices are mostly resource-constrained in CPU, memory, and storage. Therefore, the security auditor must consider a trade-off between accuracy and efficiency in designing and implementing such analysis tools.
2) **The intrinsic gap between greybox fuzzing and hardware requirement.** The COTS manufacturers installed proprietary programs on their IoT devices before shipping. As a result, all existing solutions require the emulator to fuzz those closed-source programs. However, for IoT devices, many peripheral sensing devices are not supported by emulators. This kind of conflict leads to the dilemma of applying fuzzing techniques to closed-source software on IoT devices.
3) **Fuzzing daemon programs on IoT devices.** Software on network-enabled Linux-based IoT devices commonly waits for remote network commands from other devices. Those network programs often appear as daemon programs on the device, which start just once with configured parameters. As a result, fuzzing daemon programs with standard input is infeasible since no more parameters will be further taken as input.

None of the existing works can address all these challenges. Our goal is to propose a novel approach to adapt the existing fuzzing tool to work with binary programs on IoT devices for efficient and comprehensive security analysis. In a recent survey (Pro, 0000), 71.8% of the respondents chose to use Linux, including Android and Android Things, demonstrating that Linux is still one of the most popular operating systems even for IoT devices. To this end, we design and implement AflIot, a coverage-guided greybox fuzzing framework for Linux-based IoT binary programs, which is generic, reliable, accurate, and capable of fuzzing most programs directly on devices. Because most Linux-based IoT devices use ARM instruction sets, we choose the ARMv7 devices to implement our framework.

For emulator-based approaches such as FIRMADYNE (Chen et al., 2016) and AFL (QEMU mode) (Zalewski, 0000), they require a certain amount of manual work to re-hose programs on the x86 machine. In contrast, our AflIot provides a generic and lightweight fuzzer, which instruments target binary programs statically and can be deployed directly on various Linux-based IoT devices. In this way, it becomes possible to fuzz program on IoT devices directly and, therefore, address the first challenge.

As for the second challenge, our solution consists of two phases for each target binary program in fuzzing: 1) the instrumentation phase; 2) the fuzzing phase. As shown in Fig. 1, AflIot first completes the instrumentation phase on an x86_64 server and then deploys the instrumented program with the fuzzer on the IoT device. Our two-phase design significantly eases the burden on IoT devices. Besides, AflIot leverages AFL (Zalewski, 0000), a well-known lightweight coverage-guided greybox fuzzer, and extends it by implementing binary-level instrumentation. Hence, it can support the native execution of target binary programs and reduce the performance overhead dramatically. Also, there are many existing fuzzers (e.g., QSYM (Yun et al., 2018), Driller (Stephens et al., 2016), VUzzer (Rawat et al., 2017), Firm-AFL (Zheng et al., 2019)) that claim better performance than AFL. However, they can hardly be applied for closed-source binaries due to the peripheral requirement and limited resources.

Moreover, our design also provides solutions for fuzzing on network daemon programs. AflIot can forward all inputs from a fuzzer to a specific network interface where the target daemon program listens. We achieve such forwarding through hook operations on Linux socket APIs. Hence, AflIot avoids expensive network operations, which makes it more efficient for fuzzing on network daemon programs.

To the best of our knowledge, we are the first to provide a generic and practical greybox fuzzing solution to binary programs on Linux-based IoT devices. We evaluated AflIot on benchmarks and real-world binary programs on commercial IoT devices. By fuzzing each program for 48 hours on real-world IoT devices, AflIot successfully detected 437 unique crashes among 13 binary programs on three devices, including 95 confirmed crashes from the manufacturer. The experiments demonstrate that AflIot achieves comparable performance on branches and unique crashes detection compared to AFL when fuzzing closed-source programs. AflIot is more efficient, effective, and capable of detecting bugs for binary programs on Linux-based IoT devices[3].

In summary, our work makes the following contributions:

- We present the first generic and practical fuzzing solution for binary programs on physical Linux-based IoT devices.
- We design and implement a reliable open-source fuzzing framework called AflIot by applying sophisticated binary-level instrumentation techniques.
- We integrate a network input redirection with our binary instrumentation techniques to enable generic greybox fuzzing for daemon programs on Linux-based IoT devices with limited computation resources.
- We evaluate AflIot on both benchmarks and real-world IoT devices. Our experimental results prove its reliability and efficiency in finding potential vulnerabilities in binary programs on Linux-based IoT devices.

[3] we provide our source code at https://www.github.com/SocietyMaster/AFLIoT.git

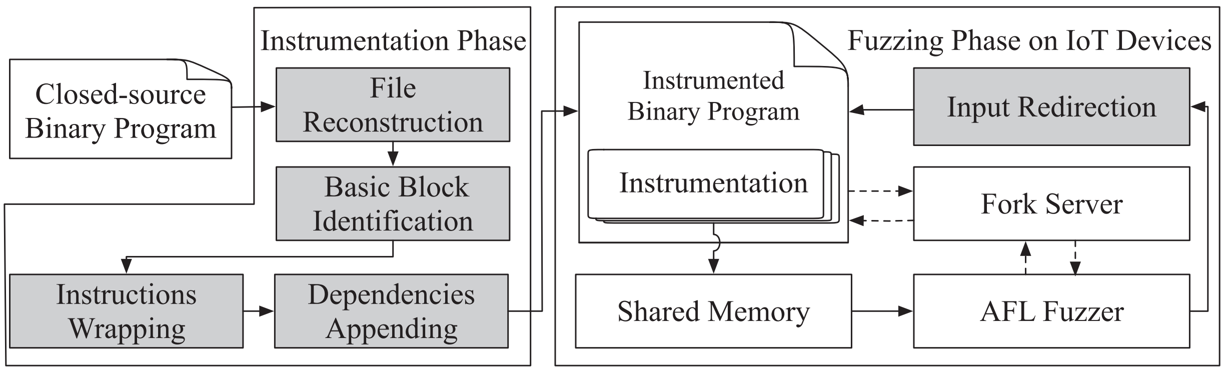


Fig. 1. Overview of AflIot. Grey blocks represent unique components in AflIot and white blocks represent the AFL's components leveraged by AflIot.

The outline of this paper is arranged as follows. We explain the necessary background in Section 2. We illustrate the design of AflIot and discuss the approach and algorithms in Section 3. We provide technical details of implementation in Section 4. We evaluate the reliability and correctness of AflIot and compare it with AFL on fuzzing closed-source binary programs in Section 5. We discuss the limitations and possible improvements in Section 6. We present the related works of AflIot in Section 7. Finally, we provide our conclusion in Section 8.

2. Background

2.1. American fuzzy lop

American Fuzzing Lop (AFL) (Zalewski, 0000) is a famous coverage-guided greybox fuzzer widely used to inspect software incorrectness in academia and industry. AFL instruments the target program, mutates the original input seed, and keeps tracking the target program's execution. The entire greybox fuzzing procedure of AFL can be divided into the **instrumentation phase** and **fuzzing phase**.

The instrumentation phase aims to insert specific instructions into programs to track access to basic blocks when executing the target program. The fuzzing phase pursues higher coverage by leveraging the instrumentation information. A higher coverage leads to more possibilities of triggering new internal states, which increases the possibility of detecting new bugs.

In the instrumentation phase, AFL chooses different instrumentation method to obtain instrumentation information depending on the transparency of the source code. AFL instruments open-source programs while compiling source code into assembly code. AFL enables itself to insert predefined instrumentation into the assembly code by patching the assembler. The patched assembler can scan transition characteristics of assembly code and determine the position where to instrument. This type of compiling-time instrumentation for open-source code is called **assembly-level instrumentation**.

Unfortunately, such instrumentation during compiling time is not available for closed-source programs. AFL uses the techniques of QEMU binary translation and QEMU patching to achieve greybox fuzzing. Since the basic block transition (such as block jump in QEMU emulator) requires the on-the-fly action between emulated devices and host, AFL tracks them by hooking the block translation and saving transition information. Since the patched QEMU only records instrumentation information when the basic block is executing, this type of instrumentation is called **emulator-level instrumentation**.

In comparison, the assembly-level instrumentation has better performance than the emulator-level one, whereas the latter has better replaced[id=xcdu]support supports for devices based on the ARM instruction set architecture. The ARM instruction set is commonly applied to Linux-based IoT devices.

In the fuzzing phase, the fuzzer first loads the initialized original seed into the queue and then obtains the next input. This procedure is called seed scheduling. Later, the fuzzer performs an input pruning step to minimize the input on the premise of not interfering with program behavior. AFL mutates the seed during the fuzzing process with predefined strategies after the input pruning. AFL mutates the seed with predefined strategies after the input pruning during the fuzzing process. Then the mutated seed is fed into the fuzzed program to trigger program functions and obtain instrumentation information. If new basic block transitions are detected, AFL will append the mutated seed to the next round of fuzzing queue.

2.2. QEMU

QEMU (qem, 0000) is a processor emulator that supports various instruction set architectures, including ARM, MIPS, x86, and x86-64. It also supports dynamic binary translation that can translate the emulated device's instructions into those on the host on-the-fly. QEMU can conduct such translation at the block level and cache each translated block for reuse. If a direct jump from a translated block is determined and the destination block is cached, QEMU will chain the translated block with its successor to avoid re-translation. Otherwise, QEMU requires a jump to the QEMU core to calculate the destination address or translate the successor block. Then the emulation resumes. This kind of block transition will result in performance costs.

Furthermore, QEMU has two operating modes: full system emulation and user mode emulation (qem, 0000). The full system emulation emulates the entire system, including the processor and its peripheral devices. The user mode emulation can launch the process of an emulated device on the host CPU. Generally, full system emulation has better compatibility for various devices than user mode emulation, while the user mode performs better than full system emulation.

These characteristics provide QEMU itself a position with a particular advantage in IoT program fuzzing, especially the greybox fuzzing. The dynamic binary translation lets the fuzzer obtain the instrumentation information when fuzzing closed-source binary programs. Note that the full system emulation provides a possible method through its expensive interface to emulate IoT devices with inevitable performance downgrade. Such performance downgrade mainly comes from QEMU fully emulating processors, hard disk input and output, or other interrupts. However, due to the enormous number of various devices and peripherals, it is impossible to fuzz programs for IoT devices by the manual customization of emulators.

3. Design

This section illustrates our system design of AflIot and discusses the approaches, algorithms, and details in AflIot.

3.1. Overview of AflIot

As illustrated in Fig. 1, there are two typical phases in the workflow of AflIot: the **Instrumentation Phase** and the **Fuzzing Phase**.

In the instrumentation phase, AflIot includes four necessary components to instrument the target binary program:

File Reconstruction rearranges the file structure of the target program. After the instrumentation, new instructions are necessarily involved. AflIot preallocates the space to store instrumented stubs, functions, and other dependencies rather than overwriting the original data. See Section 3.3 for details.

Basic Block Identification determines the address to instrument the stubs. All entry addresses of basic blocks inside the target binary file are considered to be instrumented. See Section 3.4 for details.

Instructions Wrapping wraps the instructions on the addresses already placed with stubs to avoid disrupting offsets in the target program. See Section 3.5 for details.

Dependencies Appending appends functions such as initialization, recorded branch information, libraries for network input redirection, and other dependencies. See Section 3.6 for details.

We will first introduce the overall binary instrumentation algorithm in Section 3.2 and then discuss the complex components listed above.

In the fuzzing phase, *Input Redirection* of AflIot helps the fuzzer feed the program's input during fuzzing. The non-daemon program's input is fed through standard inputs as usual, whereas the input of the daemon program is fed into a predefined port by redirections. We will illustrate the *Network Input Redirection* for daemons in Section 3.7.

3.2. Binary instrumentation algorithm

Algorithm 1 explains the detailed procedures in the correspond-

Algorithm 1 Instrumentation Algorithm.

```
1: function DoInstrumentation($path_{elf}$)
2:     $file_{elf} \leftarrow$ InputStream of $path_{elf}$
3:     ShiftAndAppendSections($file_{elf}$)
4:     $set_{instr} \leftarrow$ LocateInstructionToStub($file_{elf}$)
5:     for each $instr \in set_{instr}$ do
6:         $block_W \leftarrow$ WrappedBlock($instr$)
7:         $block_{stub} \leftarrow block_T + block_W$
8:         $vaddr \leftarrow$ AllocStubBlockAddr($block_{stub}$)
9:         ReplaceDestAddr($instr, vaddr$)
10:    end for
11:    AppendInitAndDependencies($file_{elf}$)
12:    WriteChangesToFile($file_{elf}$)
13: end function
```

ing instrumentation phase. To differentiate from the word *instruction*, which also has an *instr-* prefix, we use the *stub* to represent *instrumentation* operation in Algorithm 1. The *instr* denotes a line of the *instruction*, which is the component of a *basic block*.

Algorithm 1 targets the Executable and Linkable Format (ELF) (Contributor, 0000) binary programs because ELF is the most common file structure in Linux-based systems. We describe the structure of a typical ELF file in Fig. 2a.

When instrumenting at the binary level, directly inserting instructions into basic blocks will lead to the offset error. To avoid that, at the beginning of the instrumentation phase, we need to recheck the sections contained in the target program and reshape its structure.

Function ShiftAndAppendSections uniformly shifts the subsequent sections of the *Program Header Table (PHT)* and appends patched code and data sections. As shown in Fig. 2, PHT defines the runtime information for the binary program.

Each of those sections in the target program holds a temporary virtual address and will be finally applied when AflIot calls Function WriteChangesToFile.

Function LocateInstructionToStub identifies all possible function entries and branch instructions. AflIot takes the destination in branch instructions, the subsequent instructions after branch, and the function entries into consideration to ensure the correctness of the program functionality after instrumentation. This kind of entry instruction of basic blocks is called **boundary instruction** of basic blocks. A stub replaces a boundary instruction through ReplaceDestAddr.

Each stub can redirect execution to a basic block called **stub block**, which manages to record branch information and execute the replaced instruction. The term *block* denotes a code block that contains several lines of instructions. The $block_T$ is the **tracing block** that can always trace branch information. The $block_W$ is the **wrapped block** to execute the wrapped boundary instruction, which is abbreviated as **wrapped instruction**. Both $block_T$ and $block_W$ are composed of the intact stub block $block_{stub}$.

Function AllocStubBlockAddr is utilized to generate the preallocated address for each stub, determining the destination address in the replaced instruction after wrapping. Then, Function AppendInitAndDependencies appends necessary initialization and dependencies to the patched sections to make the program compatible with the AFL fuzzer. Finally, when no more changes are required, the entire modification is flushed into a file by Function WriteChangesToFile.

3.3. Sections arrangement

As illustrated in Fig. 2(a), *ELF Header* contains the necessary information on the file, such as the instruction set and architecture. Besides, *ELF Header* also stores the position and size for both the program and the *Section Header Table (SHT)*. SHT specifies each section's position and size within the ELF file, which is useful during the linking process.

In an ELF file, sections only contain static information. So the instrumented data and instructions cannot be loaded into the memory during execution unless we declare those entries in PHT for each new section. ELF used the declared entries to specify the access permission for each segment of data and the memory address to load.

Unlike expanding SHT located at the end of the ELF file (shown in Fig. 2a), expanding PHT will inevitably erase the sections that come after it.

To cope with that situation, we separate the entire ELF modification procedure into two different procedures:

1. ShiftAndAppendSections (Algorithm 1 Line). Each section assumes a soft offset calculated when AflIot writes changes to the target file. When shifting the sections after PHT, we only need to shift the soft offsets of each section. Similarly, when appending new sections for the patched code (section *.pcode*) and data (section *.pdata*), as illustrated in Fig. 2b, AflIot will also create two new soft offsets for those two new sections, respectively.

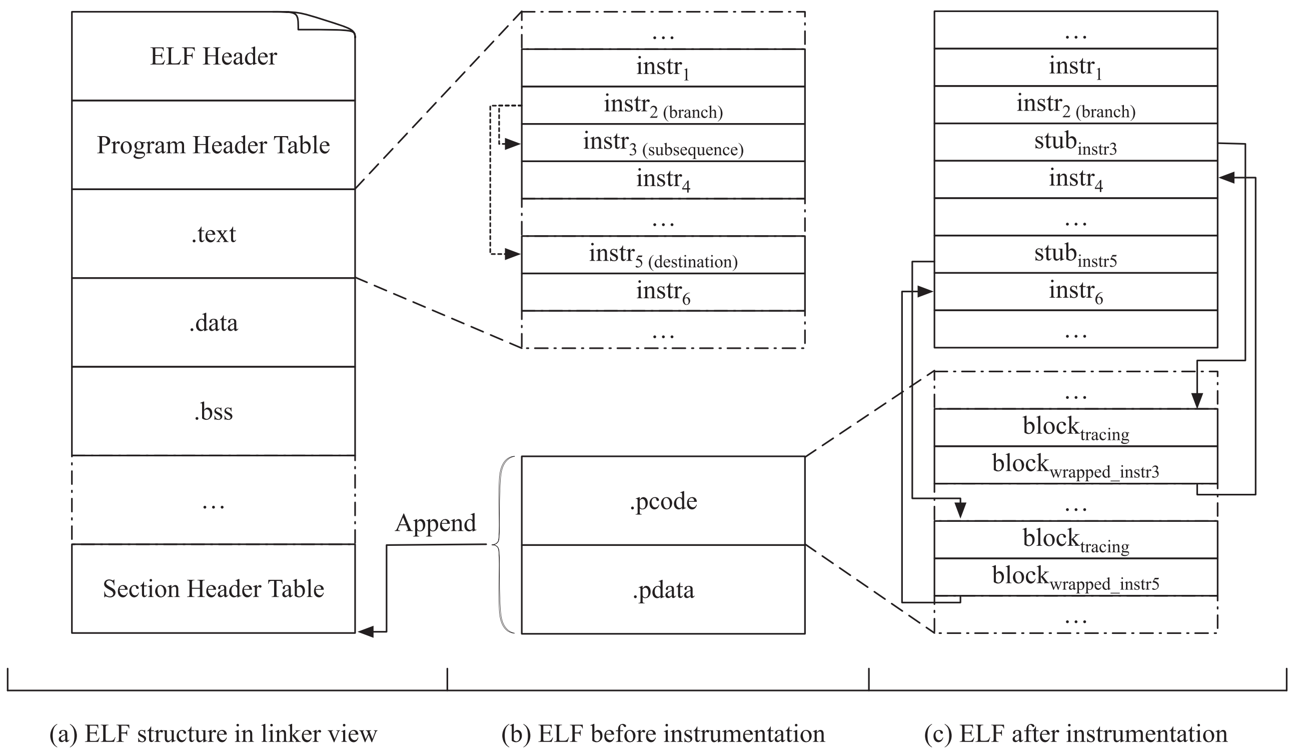


Fig. 2. ELF file instructions wrapping.

2. WriteChangesToFile (Algorithm 1 Line). When no further modification for sections is required, and the only remained task is to write changes to the file, AflIot will convert the soft offsets to the practical addresses before saving. Moreover, AflIot directly instruments the patched code and data in binary format into the target program.

3.4. Basic block identification

Coverage-guided greybox fuzzer improves the execution coverage by detecting each execution path of the ELF binary program. The fuzzer tracks basic blocks to obtain the execution path information. A basic block is a straight-line code sequence without branches (Hennessy and Patterson, 2011). To this end, AflIot necessitates accurate identification and division of the basic blocks.

As described in Algorithm 2, AflIot first statically analyzes

Algorithm 2 Locate Instructions to Instrument.

```
1: function LocateInstructionToStub($file_{elf}$)
2:     $s_{instr} \leftarrow new$ Set()
3:     $s_{instr} \leftarrow s_{instr} \cup$ FunctionEntryInstr($file_{elf}$)
4:     for each $instr \in$ Instructions($file_{elf}$) do
5:         if $instr.operation \in operations_{BRANCH4pt}$ then
6:             $s_{instr} \leftarrow s_{instr} \cup$ NextInstr($instr$)
7:             $s_{instr} \leftarrow s_{instr} \cup$ DestInstr($instr$)
8:         else if $instr \in instrs\_PC\_BRANCH4pt$ then
9:             $s_{instr} \leftarrow s_{instr} \cup$ NextInstr($instr$)
10:        end if
11:    end for
12:    return $set_{instr}$
13: end function
```

the target binary program, recording entry instructions for each function. Then, AflIot searches for PC-independent transfer instructions (Algorithm 2 Line) and PC-related transfer instructions (Algorithm 2 Line). The **PC** is the abbreviation of *Program Counter*. In the case of ARM, the PC-independent transfer instructions refer to jump instructions that do not include PC in their operands. The PC-related transfer instructions include PC-write instructions or jump instructions with PC in the operands.

For the PC-independent transfer instruction, it is the exit of a basic block, i.e., its next instruction and the destination instruction will be the entry of the other basic blocks. For PC-related transfer instruction, since in most cased its transfer destination can only be determined at runtime, AflIot only considers its next address to be entry point of other basic blocks.[4]

3.5. Instructions wrapping

To instrument a closed-source ELF program, the naive idea is to insert a function call to the beginning of basic blocks. Unfortunately, because an ELF binary program has already been compiled, it remains no extra space to instrument the stub codes directly in the *.text* section. Moreover, moving part of instructions or data to make room for instrumentation will disrupt the target program's execution due to potential wrong address offsets. As a result, such a straight-forward idea is not feasible in practice.

Note that most IoT devices are built upon RISC architectures (e.g., ARM, MIPS) with a fixed instruction length. Hence, we can leverage the so-called instruction wrapping methodology to instrument code stubs without considering the alignment issue, replacing the instruction where we need to instrument with a redirection jump to our codes, as long as we can ensure the following things:

1. AflIot can correctly record the current register states when it is about to execute the original boundary instruction.

[4] In fact, for the destination of PC-related transfer instructions, AflIot will entrust it to IDA Pro (ida, 2020) to analyze. Nevertheless, AflIot can only find as many destination addresses as possible that can be computed by the off-the-shelf static analysis tool.

2. AflIot can recover the register states from the original execution location when the wrapped block is executed.
3. AflIot does not modify any other data related to the target program except for the general-purpose registers and the stack pointer.
4. The original stack data that is not used by the stub block should not be modified.

The save-and-restore instrumentation method does not affect the jump calculation of the ELF program, so it can also support the instrumentation of the Position-Independent Code (PIC).

Wrapping Workflow. We can identify the boundary instructions of basic blocks as instructions to wrap, shown as $instr_3$ and $instr_5$ in Fig. 2(b) and 2(c). As to $instr_3$, we replace the boundary instruction with the stub instruction. The stub instruction jumps to the entry of the tracing block $block_{tracing}$, and the tracing block makes tracks for branch information. The code block $block_{wrapped\_instr3}$ that executes the boundary instruction after replacement follows the $block_{tracing}$. At the end of $block_{wrapped\_instr3}$, we append a return to the next instruction of boundary instruction (i.e., $instr_4$) to resume the execution. The $instr_5$ applies the same wrapping workflow as what $instr_3$ does. Such wrapping and instrumentation can be applied to other boundary instructions in *basic block identification*.

We instrument the stub instruction at the position of the boundary instruction, whereas we append the implementation of the instrumentation in another section (i.e., *.pcode* section). The offsets of the tracing block and wrapped block are not determined unless we finish appending the stub block. Finally, the destination of the stub instruction will be calculated when we combine the pre-arranged offset of *.pcode* and the offset of the stub block.

Type of Wrapping. In the instruction wrapping stage, the major challenge is that we should guarantee the equivalent execution of boundary instructions such as $instr_3$ and $instr_5$ in Fig. 2b.

According to (ARM, 0000), the typical ARM instruction includes the following necessary information:

$$\underbrace{(loc)}_{location}\quad \underbrace{op\{cond\}}_{operation}\quad \underbrace{r_d,\ r_n\ \{, operand2\}}_{operands}$$

An instruction is composed of an operation and one or more operands. The *operation* consists of the operation code *op* and the condition code *cond* that can conditionally execute the operation based on flags in *Application Program Status Register (APSR)*. The format of *operands* depends on the type of operation. The operands include the destination register r_d and the source register r_n in most cases. Sometimes it also includes an extra operand *operand*2. The location *loc* is the hidden information deduced from the offset of instruction, which is the essential information in the instruction wrapping.

Briefly, what we need to do is to maintain the values in all registers or memory addresses that are used in the target program. That is, when redirecting to our stub block, we need to back up all registers before using them and recover their values before return. In any case, we should only write to those unused addresses and registers to avoid potential conflicts. However, such wrapping is quite complicated when considering the practical facts. For instance, ARM instructions support conditional execution, which controls whether to execute the current instruction. What's more, some boundary instructions refer to or modify the PC's value. Since PC indicates the current address of the instruction being executed, its value varies at different places during the program execution. Wrapping those PC-related instructions in another section may break the original program's execution and lead to errors. Besides, the boundary instruction and the stub block may also involve stack manipulation. If not handled properly, it may incorrectly modify existing values in the stack. In more serious cases, it can lead to fatal errors.

To counteract those effects, we categorize all kinds of instructions into four types based on their operation code and the relationship between PC and operands, as illustrated in Fig. 3.

1) **PC irrelevant instruction** ($type_{ir}$): This type of instruction includes instructions where PC does not occur in *operands*.
2) **PC read-only instruction without push operation** ($type_{rnp}$): This type of instruction reads the value of PC in r_n or *operand*2 but does not update the value of PC after execution. The operation of this type of instruction should not be the push operation.
3) **PC read-only instruction with push operation** ($type_{rp}$): This type of instruction only includes the push operation with reading the value of PC in r_n or *operand*2. The value of the PC should not be updated by this type of instruction.
4) **PC write instruction** ($type_w$): This type includes instructions that the value of PC will be updated after the instruction is executed.

As illustrated in Fig. 3b-3 e, each replaced stub instruction is presented on top of their replaced instruction in each subfigure. Each boundary instruction with the strikeout line is followed by a stub instruction redirected to the tracing block. In Fig. 3a, for each type of boundary instructions to wrap, AflIot provides different wrap solutions.

In general, we replace the boundary instruction and jump to a new copy of the tracing block. As shown in Fig. 3(b)-3(e), the tracing block is followed by a wrapped block. Apart from the $type_{ir}$ instruction, we leverage available memory addresses that are lower than the current stack pointer (SP) to temporarily back up the registers that we are using in the wrapped block. We also utilize a register that is not involved in boundary instructions, called the **pivot**, as a substitute for PC, which can constantly store the correct value of PC and avoid being influenced by PC changing. After we back up all registers, AflIot will place a copy of boundary instruction (i.e., wrapped instruction) in the stub code and replace the PC with pivot inside the wrapped instruction. All registers except the PC will be restored before jumping back to the next destination of boundary instruction, and the PC will be automatically recovered after the jump back. Besides, we should keep the same value of SP as what it is before wrapping to fetch its value in the wrapped instruction.

For $type_{ir}$, we can execute boundary instruction inside the stub block and then jump back, as shown in Fig. 3b. For $type_{rp}$, we should separately push pivot and other registers due to pivot replacing, as shown in Figure 3 d. For $type_w$, the PC should be the last register to be restored, as shown in Fig., considering that the change of PC will lead to a direct jump.

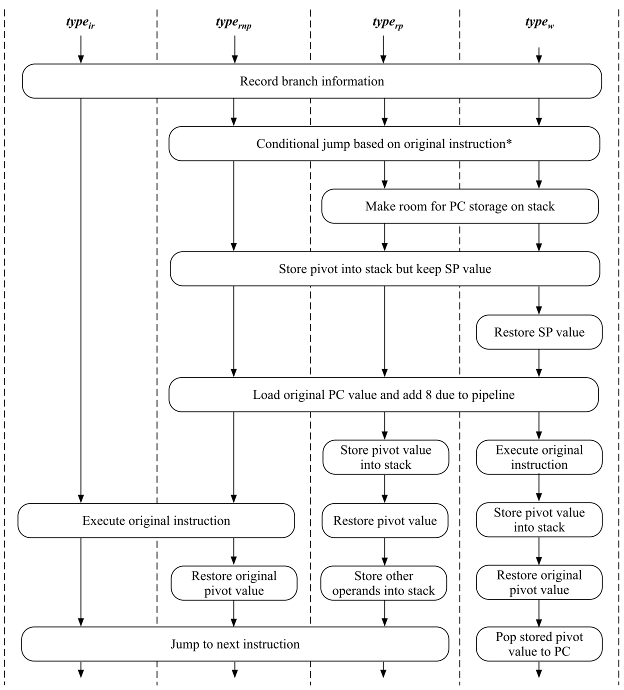
In the ARM instruction set, most instructions have their variant versions for conditions. For a boundary instruction *cond*, we insert one line of code between tracing and wrapped block, allowing it to jump to its origin conditionally, as shown below:

$$b\{cond\}\quad \underbrace{loc_{boundary\_instr}+4}_{destination\ \ location}$$

As illustrated at loc_{41030} in Fig. 3c, if the condition of instruction fails, the instrumented target program will record the branch information and directly jump out of the stub block.

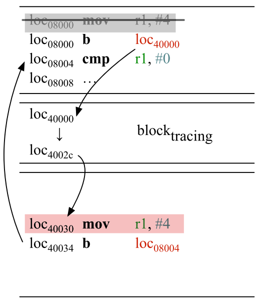
Our instrumentation can be applied to various target binary ELF files on many different Linux-based IoT devices. Moreover, our instruction wrapping methodology also has the following advantages:

- When we replace the boundary instruction to redirect to stub block, we will not disrupt other branch instructions since their offsets will not change.



$type_{ir}$
$type_{rnp}$
$type_{rp}$
$type_{w}$
Record branch information
Conditional jump based on original instruction*
Make room for PC storage on stack
Store pivot into stack but keep SP value
Restore SP value
Load original PC value and add 8 due to pipeline
Store pivot value into stack
Execute original instruction
Execute original instruction
Restore pivot value
Store pivot value into stack
Restore original pivot value
Store other operands into stack
Restore original pivot value
Jump to next instruction
Pop stored pivot value to PC

(a) Workflows for Four Instruction Wrapping Types.

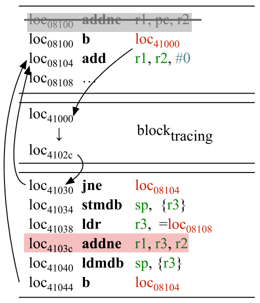


```
~~loc08000  mov    r1, #4~~
loc08000  b      loc40000
loc08004  cmp    r1, #0
loc08008  ...

loc40000
   ↓              block_tracing
loc4002c

loc40030  mov    r1, #4
loc40034  b      loc08004
```

(b) Example of $type_{ir}$.

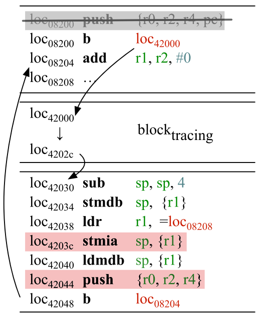


```
~~loc08100  addne  r1, pc, r2~~
loc08100  b      loc41000
loc08104  add    r1, r2, #0
loc08108  ...

loc41000
   ↓              block_tracing
loc4102c

loc41030  jne    loc08104
loc41034  stmdb  sp, {r3}
loc41038  ldr    r3, =loc08108
loc4103c  addne  r1, r3, r2
loc41040  ldmdb  sp, {r3}
loc41044  b      loc08104
```

(c) Example of $type_{rnp}$.

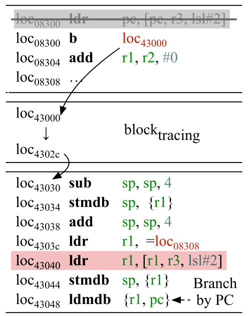


```
~~loc08200  push   {r0, r2, r4, pc}~~
loc08200  b      loc42000
loc08204  add    r1, r2, #0
loc08208  ...

loc42000
   ↓              block_tracing
loc4202c

loc42030  sub    sp, sp, 4
loc42034  stmdb  sp, {r1}
loc42038  ldr    r1, =loc08208
loc4203c  stmia  sp, {r1}
loc42040  ldmdb  sp, {r1}
loc42044  push   {r0, r2, r4}
loc42048  b      loc08204
```

(d) Example of $type_{rp}$.



```
~~loc08300  ldr    pc, [pc, r3, lsl#2]~~
loc08300  b      loc43000
loc08304  add    r1, r2, #0
loc08308  ...

loc43000
   ↓              block_tracing
loc4302c

loc43030  sub    sp, sp, 4
loc43034  stmdb  sp, {r1}
loc43038  add    sp, sp, 4
loc4303c  ldr    r1, =loc08308
loc43040  ldr    r1, [r1, r3, lsl#2]
loc43044  stmdb  sp, {r1}      Branch
loc43048  ldmdb  {r1, pc} ◂- - by PC
```

(e) Example of $type_{w}$.

. *Conditional jump will be omitted if the boundary instruction is not a conditional instruction. In Figure 3, only Figure 3c contains conditional jump instruction in loc_{41030}.

. ▭ represents the boundary instruction replaced by stub instruction. 🟥 indicates the location(s) that equivalent wrapped instruction(s) of the boundary instruction shall be executed.

Fig. 3. Workflows for each type and contrastive examples for each step in workflows.

- By avoiding instruction moving or insertion separately, we can shift all sections together with a unified offset to make room for updating PHT.
- Our strategy of appended sections is also compatible with other dependent functions, such as the AFL fork server's initialization.
- Since we focus the wrapping on the binary level, we can handle the stripped binaries.
- The save-and-restore strategy for wrapped instructions makes AFLIoT capable of PIC instrumentation.

3.6. Appending dependencies

In addition to instrument tracing and wrap blocks, AFLIoT requires to append dependencies for other essential functions. Moreover, AFLIoT should also update PHT and other critical sections to ensure that those appended dependencies will be loaded appropriately.

3.6.1. Initialize fuzzer

For coverage-guided fuzzing, the initialization of the target program will launch a proxy called fork server to enable communication between fuzzer and target program through mapping shared memory address. The fork server is also responsible for continuously forking the target program to improve the coverage.

Generally, the initialization should be instrumented before the start of the target program, but sometimes locating the entry point for instrumentation is quite complicated. For robustness, we leverage the *.init_array* section in an ELF file to execute initialization instructions in the target program. The *.init_array* is where each element specifies a function to be executed at the beginning of the execution. However, moving the *.init_array* somewhere else or appending entries to its end is not that simple. Its length is the hard-coded in caller function. Thus changing such length will require reassembling.

AFLIoT uses a different approach to initialize the fuzzer. We replace one entry already in added[id=xcdu]the *.init_array* with a stub redirecting to a function that performs the same as the caller does, except that it will pick initialization entries from another array. Both the new entries added to initialize the fork server and the old entries replaced are included in that array. This solution is adaptive to the ELF executables and shared libraries. Such initialization may not be limited only to the fork sever. It can be extended to other applications, such as network input redirection for the daemon.

3.6.2. Update program header table

PHT will be updated at last because AFLIoT should know how much those subsequent sections should be shifted. Since we already wrapped boundary instructions at this moment, AFLIoT should first update all the address holders used in instruction wrapping before and then the PHT, ensuring all essential dependencies can be loaded during fuzzing.

3.7. Network input redirection

In traditional x86 devices, most closed-source binary programs that need to be fuzzed will receive either a local file or a standard I/O stream. However, since IoT devices pervasively require connectivity to other COTS devices, it is not feasible to fuzz network-required daemon binary programs by feeding inputs through such two manners. Binary daemons listen on specific ports and interact with clients from other devices. To fuzz on IoT devices directly, AFLIoT needs to appropriately treat those network-required binary programs appropriately.

As illustrated in Fig. 4, AFLIoT leverages redirection from standard input to network socket to forward inputs generated by fuzzer. The network input redirection comprises fuzzer, instrumented daemon program, and input redirection. The fuzzer is the

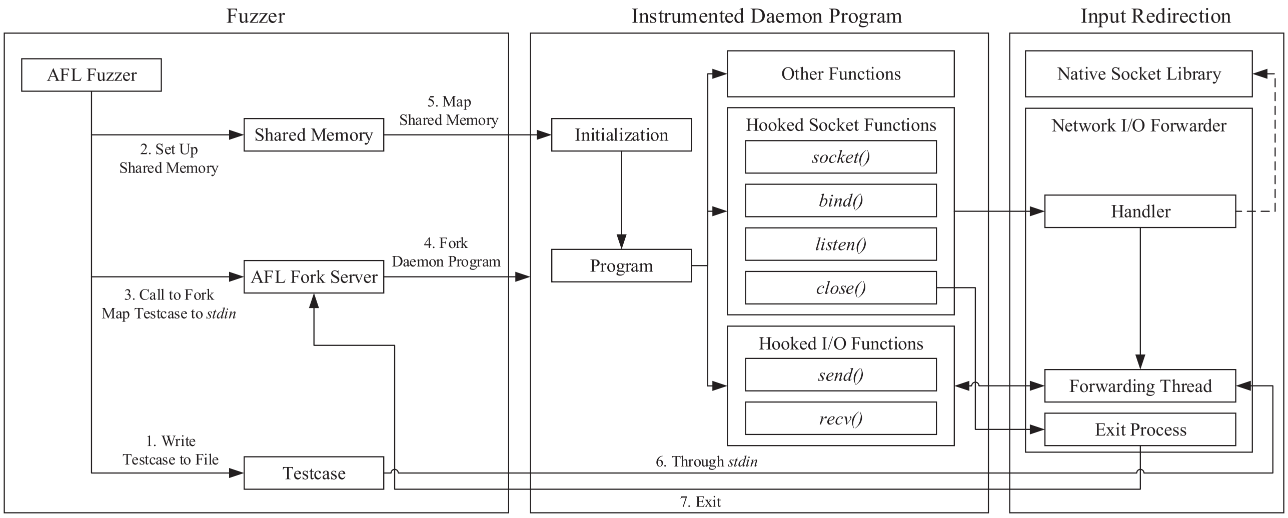


Fig. 4. Daemon program lifecycle and network input redirection.

normal AFL fuzzer without any modification. The critical component is the input redirection on the left side of Fig. 4. It hijacks standard input from the fuzzer. When the fuzzer generates mutated input, the input redirection will continuously forward the input to a predefined port, which means the target program fuzzed can reach deeper branches without hanging up. In Fig. 4, AflIot hooks three types of network-related API.

- Basic socket APIs, including *socket, bind, listen*, and *accept*, create pthread to receive incoming contents from the network.
- I/O multiplexing APIs, including *select, poll*, and *epoll*. When daemon programs call those APIs, AflIot will forward the standard input into a predefined network port.
- Finalizer API, i.e., *close*, terminates the pthread that forwards the content to a network port.

Figure 4 also shows the entire lifecycle of network input redirection and fuzzed program, which starts with the network event simultaneously and ends when the network event is closed. Since AflIot redirects standard inputs, it intrinsically inherits the capability of mutation from those coverage-guided fuzzers.

4. Implementation

In this section, we describe the technical details of implementing AflIot.

4.1. Device setup

The programs are closed-source on most Linux-based IoT devices, and the devices are not accessible by default for security reasons. We ensured the following to enable the on-device grey-box fuzzing through AflIot.

Access devices externally. Most Linux-based IoT devices operate with the firmware that contains secure shell and telnet, or the firmware is somehow compatible with them. For example, Raspberry Pi4 and Xiaomi Router provide remote access interfaces officially. Hence, we can enable our privilege of secure shell and leverage it to access the target program.

Some IoT devices support remote access but do not provide explicit interfaces to enable it. Manufacturers may embed remote access in devices for debugging, which should not be accessed by regular users and can only be activated by secret backdoors. Therefore, it is in demand to explore such backdoors to access the shell. For instance, by sending a specially crafted packet, the telnet service of Netgear Routers will be enabled.

Some firmware shipped with devices provides neither remote access service nor the interface to enable such service. However, the capability of flashing is still enabled. Thus, we can flash them with specifically modified versions of firmware with remote access features for those kinds of devices.

Otherwise, we have to use other physical methods to establish a connection. For instance, in an ASUS router, we leveraged its UART serial port to access the console and enable secure shell service.

Once the remote access interface is enabled, the console and file system accesses are also available. We can transfer the fuzzer, the instrumented target programs, and other necessary dependencies to devices and start the fuzzing process.

Privilege to write data. We have to store our fuzzer and instrumented program on the device. However, sometimes the storage on devices may not be writable. In this situation, both external storage and internal memory are good options for storage. Specifically, we used external storage on the Raspberry Pi4 and those routers to perform our fuzzing.

4.2. Toolchain

In the instrumentation phase, we leverage the *FlowChart* function of IDA Pro (ida, 2020) to identify the functions and basic block information of target programs, as aforementioned in Section 3.4.

Then we leverage Capstone (Quynh, 0000) to disassemble entry instructions of each basic block. Finally, when writing back changes into the file, we utilize Keystone (key, 0000) to assemble modified instructions and data into the original binary file.

4.3. Fuzzer integration

Classical AFL retrieves the execution information of a target program by shared memory. Specifically, AFL sets up a shared memory before forking the target program and then maps it into his own memory space. As the program executes, those instrumented codes will be responsible for recording basic block transition. AFL then analyzes the transition information in the shared memory to assess the value of this test case.

The target binary should map the shared memory before executing the other parts. So we implemented a shared object for shared memory mapping. During instrumentation, the shared object dependency is injected into the dynamic section of the target binary and will be called at the very beginning by an initialization function that we inserted.

We adopt AFL's approach to identify the basic block and trace basic block transitions in terms of tracing. Each basic block is assigned with a random magic number as block ID, hardcoded in its instrumentation. Moreover, the basic block transition ID is calculated as the same as the native AFL. Each instrumentation resides at the entry of a basic block, taking the responsibility of storing the basic block transition information into the shared memory.

4.4. Shared library

Both executable programs and shared object libraries follow the ELF format in the Linux-based system. Furthermore, thanks to IDA (ida, 2020), Capstone (Quynh, 0000), and Keystone (key, 0000), AflIot's binary-level instrumentation algorithm can also function on the shared libraries.

For instance, the shared library *libthrift* is leveraged by *plugincenter* in Section 5.2. We can extract the shared library *libthrift* from the firmware, then instrument it and reload the instrumented shared library through the *LD_PRELOAD* (lds, 0000) mechanism.

4.5. Input redirection

We implement a simple input redirection mechanism between a fuzzer and a target network daemon program.

We rewrite a shared library that includes all of the standard Linux socket APIs (See Section 3.7). Through *LD_PRELOAD*, we successfully hook all these socket APIs to execute the network daemon programs and perform the parameter checks. That determines whether to intercept or forward network traffic. As shown in Fig. 4, our implementation does not require integrating our hooked library into the target program with other dependencies such as initialization and branch tracing. No further modifications are required to the target program. In this way, expensive network operations at the OS level are successfully avoided, and input redirection efficiency is improved significantly. We also want to claim that *preeny* (pre, 0000) is a more powerful framework, and it can also realize the network input hooking functionality. It can also hook the modified APIs through the *LD_PRELOAD*. However, AflIot's implementation is sufficient to cover all the Daemon lifecycle APIs as *preeny* does. So we still use our version of the implementation.

5. Evaluation

We performed plenty of greybox fuzzing experiments to validate AflIot on benchmarks and real-world IoT devices. Our primary focus is to prove the reliability and correctness of our proposal. Therefore, in this section, we mainly compare AflIot with AFL, which is the basis for many other fuzzer, such as (Böhme et al., 2017a; 2017b; Lemieux and Sen, 2018). Further enhancement from fuzzers to AFL might also be adopted to AflIot, which will be discussed in Section 6.

5.1. Instrumentation validation

To validate the correctness of the AflIot instrumentation strategy, we recorded the sequential basic block addresses of several programs fuzzed by AflIot and AFL QEMU mode, respectively. Then, as listed in Table 1, we compared the address records between AflIot and AFL QEMU mode. Furthermore, we manually reviewed and confirmed the difference according to the control flow of those target programs.

We visualize the difference gathered from the *gzip* program to illustrate the correctness of AflIot. As illustrated in Fig. 5, the entire Bar 0 represents the joint basic block address records from both AflIot and AFL. Each vertical line with different colors represents an address record of a basic block. The lines in white mean that the basic block address records from both AflIot and AFL are the same and well-aligned. The green and red lines represent the basic block addresses recorded by either AflIot or AFL (in QEMU mode). If a green and a red line are adjacent, they will be colored yellow.

Table 1
Basic block address records summary.

Program	White*	Green*	Red*	Yellow*	Total
gzip	11,654	877	9	1	12,541
bunzip2	7420	462	51	2	7935
bzcat	7409	461	51	2	7923
lzcat	9729	1864	0	0	11,593
unlzma	9763	1863	0	0	11,626
Total	45,975	5527	111	5	51,618

* **White**: it represents the aligned and same basic block address records. **Green**: address is recorded by AflIot only. **Red**: address is recorded by AFL only. **Yellow**: the context of this address can be aligned but AflIot and AFL record two conflict addresses.

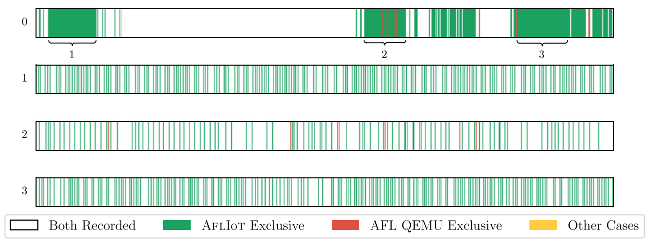


Fig. 5. *gzip* basic block address records comparison.

```
1  0x4cdac    ldr r3, [sp, #0xc]
2  0x4cdb0   loc_4cdb0:
3  0x4cdb0    ldr r5, [sp, #0x14]
4  ...        ...
5  0x4cdc4    bne loc_4cd94
6  ...        ...
7  0x4cea0    b loc_4cdb0
8  ...        ...
9  0x4d124    b loc_4cdb0
```

Fig. 6. Sample to trigger AflIot exclusive record in *gzip*.

If many red and yellow lines appear somewhere, critical instrumentation errors have occurred in that places. Due to the limited resolution in Fig. 5, those continuous regions marked as 1,2,3 in the Bar 0 are discrete basic block address records, enlarged as shown in Bar 1 to 3 correspondingly.

Table 1 summarizes the records differences between the two fuzzers. There are 12,541 basic block address records, including 11,654 white records, 877 green records, nine red records, and one yellow record. As to all the green records, the failure can be attributed to the disadvantage of the dynamic instrumentation, which relies on the execution of branch instructions to identify basic blocks.

In Fig. 6, we take one piece of instruction from *gzip* as the sample to illustrate what happened in the green lines' position. The instructions in Line 2 to 5 (from 0x4cdb0 to 0x4cdc4) should be considered one basic block because they start with a label and end with a branch instruction. However, the dynamic instrumentation in the QEMU emulator cannot determine whether the instruction in Line 2 is an entrance of a basic block when it executes to 0x4cdb0. Only when the QEMU emulator hits the branch instruction at Line 7 or Line 9 can it realize that the address 0x4cdb0 at Line 2 is the entry of the basic block. This characteristic causes the AFL QEMU mode to record basic block addresses incompletely,

Table 2
Greybox fuzzing results between AflIot on real-world linux-based IoT Devices and AFL on QEMU.

No.	Program	Type	**AflIotDevice**	Fuzzing Time(H)	Branch Coverage			Unique Crashes		
					AFL	**AflIot**		AFL	**AflIot**	
1	base64	LAVA-M	Raspberry Pi 4	48	666	733	▲ 67 (+10.06%)	0	1	▲ 1
2	md5sum	LAVA-M	Raspberry Pi 4	48	889	1116	▲ 227 (+25.53%)	0	0	
3	uniq	LAVA-M	Raspberry Pi 4	48	415	480	▲ 65 (+15.66%)	1	12	▲ 11 (+110.00%)
4	who	LAVA-M	Raspberry Pi 4	48	5035	5781	▲ 746 (+14.82%)	0	5	▲ 5
5	gunzip	Command Line	Raspberry Pi 4	48	746	768	▲ 22 (+2.95%)	51	54	▲ 3 (+5.88%)
6	gzip	Command Line	Raspberry Pi 4	48	752	780	▲ 28 (+3.72%)	51	61	▲ 10 (+19.61%)
7	lzcat	Command Line	Raspberry Pi 4	48	421	423	▲ 2 (+0.48%)	20	21	▲ 1 (+5.00%)
8	unlzma	Command Line	Raspberry Pi 4	48	424	427	▲ 3 (+0.71%)	21	22	▲ 1 (+4.76%)
9	unzip	Command Line	Raspberry Pi 4	48	1857	2016	▲ 159 (+8.56%)	5	112	▲ 107 (+2140.00%)
10	zcat	Command Line	Raspberry Pi 4	48	740	764	▲ 24 (+3.24%)	50	54	▲ 4 (+8.00%)
11	dropbear	Daemon	ASUS ACRH17	48	275	1788	▲ 1513 (+550.18%)	0	0	
12	plugincenter	Daemon	Xiaomi R1D	48	/	4851	▲ 4851	/	95	▲ 95
13	himan	Daemon	Xiaomi R1D	48	/	517	▲ 517	/	0	
Total Crash								199	437	▲ 238 (+119.60%)

which will further lead to many differences in the branch information. In contrast, the binary instrumentation of AflIot will not miss those records.

The red records originate from the variance between the basic block and QEMU Translation Block. The QEMU mechanism restricts the QEMU Translation Block transition. The restrictions include: (1) the number of translated opcodes in the translation buffer and the number of instructions in a QEMU Translation Block should not exceed a certain limit; (2) all instructions in a QEMU Translation Block should stay in a single memory page. However, the basic block may not follow those restrictions. If a block is too large to be translated by QEMU at once, QEMU will stop translating the block and split it into smaller ones. The extra split blocks are not the actual basic blocks, and they will result in the red records since the first address is not an entry. In comparison, our instrumentation implementation can track the control flow more accurately.

We note that all the yellow records listed in Table 1 are the combinations of the red and the green ones. It also explains why there are much fewer yellow records than other types.

In summary, instrumentation of the AflIot outperforms AFL QEMU mode in terms of the accuracy and correctness of the closed-source binary program instrumentation, which can be considered a reliable instrumentation methodology for fuzzing.

5.2. Evaluation on real-world IoT devices

To illustrate the correctness of AflIot, we performed greybox fuzzing on real-world devices, including Raspberry Pi4 and routers such as Xiaomi R1D and ASUS ACRH17. The Raspbian Buster on Raspberry Pi4 is with 4.19.57-v7l+ Linux kernel. Xiaomi R1D has deployed the official MiWiFi firmware (v2.25.213) with a 2.6.36 Linux kernel. ASUS ACRH17 has deployed the official firmware (v3.0.0.4.382_11812) with a 3.14.77 Linux kernel. All devices and firmware support the ARMv7l instruction set.

In comparison, we also performed greybox fuzzing through AFL QEMU user mode on a server with 256GB memory and 48 cores (Intel Xeon E5-2650 v4 2.20GHz CPU). On that server, we emulate Linux-based firmware with ARMv7l instruction set in QEMU user mode.

We need to run AflIot on a dataset as the ground truth to validate if it can correctly fuzz a program and find more bugs. Therefore, we choose AFL as the counterpart and the two most important metrics for comparison. One is the number of *Basic Block Transitions* covered (also called branch coverage for on-device fuzzing), which indicates the proportion of branches executed in the fuzzing process. The other is the number of unique crashes, which indicates the possible bugs to be identified.

Table 2 lists the test suites, including the LAVA-M dataset and several non-daemon and daemon binary programs. LAVA-M (Dolan-Gavitt et al., 2016) is a widely-used fuzzer benchmark that automatically injects a large number of bugs into four GNU core utilities programs. However, since the programs in LAVA-M are too simple, we have added several programs for further evaluation. Those non-daemon and daemon programs are deployed in the real-world COTS IoT devices.

Besides, we performed ten rounds of fuzzing on each LAVA-M program, non-daemon program, and daemon program, respectively. All of the programs were fuzzed for 48 hours in each round. The average number of basic block transitions (branch) coverage and unique crashes are shown in Table 2.

We should note that the *plugincenter* and *himan* are closed-sourced and extracted directly from Xiaomi R1D. All the open-sourced binary programs are used to compare the validity and the correctness between AflIot and AFL. We also fuzzed the daemon program to evaluate the robustness of fuzzing various closed-source programs on the network-enabled and the peripheral-dependent devices in the real world. We use the same seeds for both AflIot and AFL to ensure they have the same input when we start fuzzing.

LAVA-M Programs. We selected four classical test cases in LAVA-M for our experiment. They are *base64, md5sum, uniq*, and *who*. Each case is executed with two instances: either with AflIot or with AFL for comparison. Each instance repeats 10 times to get the average.

In the experiment, we first compiled the programs from the source codes and then considered the outputs as closed-source binary programs. AflIot performed binary-level instrumentation to the compiled binary program. In comparison, AFL performed an emulator level instrumentation during the fuzzing process.

The results in Table 2 indicate that among programs of LAVA-M, AflIot has covered more branches than AFL within 48 hours of fuzzing. For example, as for *base64*, our AflIot covers 733 branches, while AFL only covers 666, indicating an improvement of 10.06% percentage. More attention should be paid to unique crashes. For two cases (*md5sum* and *who*), AFL can not find any crash. However, our AflIot can find one and five crashes, respectively. This fact shows that AflIot can reach some code regions that AFL never touches. For *uniq*, the number of unique crashes is 11 times more than that in AFL.

Daemon Programs. In Table 2, we select daemon program *plugincenter, himan*, and *dropbear* for both Xiaomi X1D and ASUS ACRH17. Specifically, *plugincenter* and *himan* require peripheral devices, but *dropbear* does not. Since we cannot deploy *plugincenter* and *himan* into QEMU emulated host, we do not have the result of those two programs for AFL.

Table 2 shows that AflIot successfully finds unique crashes on real-world IoT devices. For the case of *plugincenter* on Xiaomi R1D, 95 crashes are found. Because we can not access the source code of *plugincenter*, those crashes were reported to Xiaomi Security Center as potential denial of service attacks, and they confirmed those vulnerabilities.

The *plugincenter* requires initialization to confirm that it communicates properly with the manufacturer and its components, such as the WiFi and Bluetooth modules. In our experiments, we have tried to fuzz plugincenter using AFL, but due to the above limitations for those closed-source binaries, we could not fuzz it directly using only the native AFL framework.

Regarding branch coverage for *dropbear*, the branch number is increased to 1788 for AflIot compared to 275 for AFL, which is about a 5.5 times increase. It indicates AflIot can access more branches by network input redirection used.

In summary, considering the LAVA-M dataset, non-daemon binary programs, and daemon binary programs, the fuzzing results indicate that AflIot is comparable to AFL on branch coverage and crashes detection. We can also conclude that AflIot provides a generic, reliable, accurate greybox fuzzing solution on IoT devices.

Note that, in this paper, we do not compare AflIot with other fuzzers such as (Böhme et al., 2017a; 2017b; Lemieux and Sen, 2018). The reason behind this can be explained as follows. First, AflIot does not involve any modification to the fuzzing strategy of the original fuzzer of AFL. Both AflIot and AFL have the same fuzzing strategy. Second, the fuzzers leveraged by (Böhme et al., 2017a; 2017b; Lemieux and Sen, 2018) have some improvements on the fuzzing strategy. The fuzzing strategy improvement is not the focus of our paper.

6. Limitations and improvements

Although AflIot has been proved a generic fuzzing tool for Linux-based IoT devices with good efficiency and accuracy, we still acknowledge the following limitations:

Variant Fuzzing Efficiency on IoT devices. Because the execution speed of the fuzzer depends on CPU frequency, fuzzing on low-performance devices could be effectively slow. Moreover, some crashes or ill-formed codes can lead to the device's instability, which makes the on-device fuzzing progress interrupted. As to that matter, this is a disadvantage of on-device fuzzing for many IoT devices. The solution to that situation includes: (1) Leveraging the persistent mode to accelerate fuzzing. The persistent mode is a singleton mode to avoid fuzzed program re-initialization by the fork server. It significantly reduces CPU and memory consumption in the forking procedure. However, we find it may occasionally lead to instability in path reproduction. (2) Leveraging distributed fuzzing. By the distributed master-slave mode of fuzzer, we can fuzz the same binary program on multiple devices simultaneously. It can accelerate the fuzzing progress at the cost of more computational resources. (3) Attempting the re-hosting approaches. The device instability caused by the fuzzed program is a critical problem for on-device fuzzing. For x86 platforms, we can manually hook or rewrite the relevant function to continue the fuzzing progress. Thus, re-hosting can be a more generic diagnosis manner for this situation.

Lack of Strategy Improvement in Fuzzing Phase. Our focus is to provide a generic solution for greybox fuzzing on Linux-based IoT devices. Thus, we construct our greybox fuzzing based on the AFL. The current version of AflIot does not involve any other strategy improvement in the fuzzing phase. If we integrate more fuzzing strategies into our solution, there should be a certain enhancement. Fortunately, the state-of-the-art improvements in greybox fuzzing enhance the efficiency of the seed scheduling and mutation. Most of those improvements, such as (Böhme et al., 2017a; 2017b; Gan et al., 2018; Lemieux and Sen, 2018), are based on AFL fuzzer. What they have improved is not conflict with the binary level instrumentation and the network input redirection of AflIot, which means AflIot can adopt those improvements further.

Incompleteness for Static Binary Instrumentation. AflIot encounters challenges such as anti-disassembly sequences, dynamically generated code, indirect branches, and shared libraries when instrumenting at the binary level (D'Elia et al., 2019). For indirect branches, as we discussed in Section 3.4, AflIot attempts to cover as many accurate indirect branch entries as possible based on the instruction type. However, the identification is still limited by the state-of-the-art disassembly techniques and static analysis. So does the identification of the basic blocks. For shared libraries, as we discussed in Section 3.5, AflIot supports the instrumentation of PIC sequence. However, the number of tested cased of shared libraries is still limited. Finally, for anti-disassembly sequences and dynamically generated code, AflIot leverages state-of-the-art techniques to analyze and disassemble the ELF files; AflIot will meet the same challenges as other works from those two factors.

Memory Limitation for On-Device Address Sanitizer. Technically, AflIot's toolchain implementation can be compatible with RetroWrite's (Dinesh et al., 2020) methodology to integrate address sanitizer. However, in practice, the memory capacity of the IoT device cannot afford the memory requirement by both the address sanitizer and AFL fuzzing framework. The QASan (Fioraldi et al., 2020) presents heap memory sanitization for binary fuzzing based on QEMU. It relies on in-emulator shadow memory to reduce the memory pressure. The QASan does not solve the memory limitation issue for on-device fuzzing, but it provides an alternative approach to sanitize binary programs if the emulator can rehost the firmware.

7. Related works

To improve seed selection for mutation-based fuzzers, Böhme et al. present the AFLFast (Böhme et al., 2017b). It uses Markov Chain to identify low-frequency paths and focuses most of its efforts on these paths because it is more likely to trigger bugs when fuzzing on these paths. Besides, Böhme et al. also introduce the directed greybox fuzzing, AFLGo (Böhme et al., 2017a), with a simulated annealing-based power schedule that assigns more energy to those seeds closer to the target location. Similar to AFLGo, Hawkeye (Chen et al., 2018a) also emphasizes the challenge of directed fuzzing, prioritizing seeds, and mutating based on the static analysis and execution trace to achieve better assistance. VUzzer (Rawat et al., 2017) leverages the control-flow graph to select input. FairFuzz (Lemieux and Sen, 2018) biases the mutation to generate inputs that can hit those branches rarely visited. Those works aim to improve seed mutation and scheduling based on AFL. However, they do not provide a solution for closed-source or peripheral devices required programs on IoT devices, which are commonly hard to migrate or be executed on QEMU.

Several works have made efforts to perform the automated security assessment on IoT devices to mitigate the risk of various attacks against IoT devices.

Costin et al. (2014) present the first large-scale analysis of firmware images with static analysis techniques. They build a system to collect much firmware from various device vendors and unpack all firmware images into millions of files. They reveal a lot of vulnerabilities, some of which are even shared by many different devices. However, their analysis still suffers from a limitation of accuracy due to their static analysis approach. Zaddach et al. (2014) propose a hybrid approach by leveraging both emulator and physical devices to perform dynamic analysis. In their approach, firmware instructions are executed inside the emulator, but all I/O operations are channeled to physical devices.

This approach enables the full emulation of the target device. However, its execution switching between emulator and physical device is quite expensive and thus lacks scalability. Besides, their implementation is specific to certain devices, so it is not easy to adopt their method to many different devices. Chen et al. (2016) present a dynamic security analysis framework for Linux-based IoT devices. First, it collects firmware from various vendors. Then, it unpacks and configures the firmware to run in an emulator automatically. Finally, it performs large-scale blackbox testing with web exploitation on the emulator. However, it is impossible to perform a comprehensive security analysis with web exploitation. Moreover, it is a non-trivial task to resolve software dependence and hardware configuration issues to prevent the kernel panic during the emulation. Their approach does not address these issues quite well, making it not applicable to the firmware. Moreover, Chen et al. (2018b) also provide IoTFuzzer, a blackbox fuzzer on the real device. It manages to perform fuzzing aimed at IoT devices by analyzing its companion mobile app to figure out the protocol. Then generate the test cases and feed them through the network. As a result of that, the throughput is slow.

Zheng et al. (2019) provide a more efficient solution, FIRM-AFL, for fuzzing IoT programs by combining system-mode emulation and user-mode emulation. They used multiple server programs as IoT programs for evaluation. However, regarding the programs on IoT devices that require peripheral devices data, they still can not provide a general solution. Gustafson et al. (2019) present Pretender, a system to automatically build hardware models. Pretender leverage MMIO to gather communication, interruptions, and peripheral states. It also uses a machine learning model to simulate interactions. It provides a proven method for modeling peripheral devices in re-hosting, but it can only simulate the model based on existing data and cannot fully interact and evaluate the security of a real peripheral device.

Song et al. (2019) present PeriScope, a Linux kernel-based probing and fuzzing framework to analyze interactions between devices and drivers. It uses MMIO and DMA mechanisms to monitor traffic between device drivers and corresponding hardware devices. Based on the PeriScope system, they found several vulnerabilities on Wi-Fi drivers. They do not improve on the instrumentation algorithm itself, but their approach of using MMIO and DMA to obtain the necessary information and implement fuzzing is enlightening. Dinesh et al. (2020) present RetroWrite, leveraging static instrumentation to fuzz and sanitize the COTS binaries. They used the symbolization to solve the instrumentation problem for x86_64 binaries and enable fuzzing and sanitization for those closed-source binaries. However, the RetroWrite requires symbols and can not handle the stripped binaries, which always exist in IoT devices. Unfortunately, the state-of-the-art method to symbolize stripped binaries is mostly heuristic, and plenty of fragmented unsymbolized basic blocks still exist in our experiments. Therefore, symbolization is not an appropriate solution for fuzzing IoT devices. Nagy et al. (2021) discuss the quality of binary instrumentation and present Zafl. It leverages binary rewriting to binary-only fuzzing with compiler-level instrumentation's capability and performance. It requires intermediate representation (IR) for rewriting and further optimization. The Zafl reaches remarkable fuzzing results and performance on x86-64 C/C++ binaries. However, due to the prevalent hard-coded offset and captious runtime environment of COTS binary programs, the IR and Zafl's optimization cause too much perturbation to binary programs and are not suitable for on-device fuzzing.

Overall, none of the existing work provides comprehensive security analysis capability with fuzzing techniques directly on IoT devices. To the best of our knowledge, we are the first to provide a practical on-device fuzzing solution to find more bugs that have not been covered or exploited previously.

8. Conclusion

In this article, we present AflIot, a coverage-guided greybox fuzzing framework for Linux-based IoT devices. To our best knowledge, AflIot provides the first practical fuzzing solution for binary programs on the Linux-based IoT device. With binary-level instrumentation, AflIot supports native execution of the target program and, therefore, intrinsically supports fuzzing IoT binary programs that require access to peripherals. We evaluate AflIot on both benchmarks and real-world IoT devices. Overall, AflIot can identify 437 unique crashes in 13 binary programs on three kinds of devices, and the manufacturer confirmed all the 95 crashes from *plugincenter*. Our evaluation shows that AflIot is efficient and effective in finding vulnerabilities and bugs in closed-source binary programs on Linux-based IoT devices.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Xuechao Du: Conceptualization, Methodology, Software, Formal analysis, Investigation, Writing – original draft, Writing – review & editing. **Andong Chen:** Data curation, Writing – original draft, Visualization. **Boyuan He:** Conceptualization. **Hao Chen:** Supervision. **Fan Zhang:** Supervision, Writing – review & editing. **Yan Chen:** Project administration, Supervision.

References

Alves, P.,. Building GDB and GDBserver for cross debugging. https://sourceware.org/gdb/wiki/BuildingCrossGDBandGDBserver.

ARM,. ARM Instruction Set Version 1.0 Reference Guide. https://static.docs.arm.com/100076/0100/arm_instruction_set_reference_guide_100076_0100_00_en.pdf.

Bastani, O., Sharma, R., Aiken, A., Liang, P., 2017. Synthesizing program input grammars. In: ACM SIGPLAN Notices, Vol. 52. ACM, pp. 95–110.

Böhme, M., Pham, V.-T., Nguyen, M.-D., Roychoudhury, A., 2017. Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 2329–2344.

Böhme, M., Pham, V.-T., Roychoudhury, A., 2017. Coverage-based greybox fuzzing as Markov chain. IEEE Trans. Software Eng..

Cadar, C., Dunbar, D., Engler, D.R., et al., 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, Vol. 8, pp. 209–224.

Cha, S.K., Woo, M., Brumley, D., 2015. Program-adaptive mutational fuzzing. In: Security and Privacy (SP), 2015 IEEE Symposium on. IEEE, pp. 725–741.

Chen, D.D., Woo, M., Brumley, D., Egele, M., 2016. Towards automated dynamic analysis for linux-based embedded firmware. NDSS.

Chen, H., Xue, Y., Li, Y., Chen, B., Xie, X., Wu, X., Liu, Y., 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, pp. 2095–2108. doi:10.1145/3243734.3243849.

Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W.C., Sun, M., Yang, R., Zhang, K., 2018. IOTFUZZER: discovering memory corruptions in IoT through app-based fuzzing. NDSS 2018, Network and Distributed Systems Security Symposium, 18–21 February 2018, San Diego, CA, USA.

Contributor, W.,. Executable and Linkable Format. https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

Costin, A., Zaddach, J., Francillon, A., Balzarotti, D., Antipolis, S., 2014. A large-scale analysis of the security of embedded firmwares. In: USENIX Security Symposium, pp. 95–110.

Def con 24 - jmaxxz - backdooring the frontdoor. http://www.co.tt/files/defcon24/DEFCON-24-Program.pdf.

D'Elia, D.C., Coppa, E., Nicchi, S., Palmaro, F., Cavallaro, L., 2019. SoK: using dynamic binary instrumentation for security (and how you may get caught red handed). In: Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, pp. 15–27.

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

Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., Whelan, R., 2016. Lava: Large-scale automated vulnerability addition. In: Security and Privacy (SP), 2016 IEEE Symposium on. IEEE, pp. 110–121.

Fioraldi, A., D'Elia, D.C., Querzoni, L., 2020. Fuzzing binaries for memory safety errors with QASan. In: 2020 IEEE Secure Development (SecDev). IEEE, pp. 23–30.
Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., Chen, Z., 2018. CollAFL: path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 679–696.
Ganesh, V., Leek, T., Rinard, M., 2009. Taint-based directed whitebox fuzzing. In: Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, pp. 474–484.
Godefroid, P., Klarlund, N., Sen, K., 2005. DART: directed automated random testing. In: ACM Sigplan Notices, Vol. 40. ACM, pp. 213–223.
Godefroid, P., Levin, M.Y., Molnar, D., 2012. SAGE: whitebox fuzzing for security testing. Queue 10 (1), 20.
Godefroid, P., Peleg, H., Singh, R., 2017. Learn&fuzz: machine learning for input fuzzing. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, pp. 50–59.
Gustafson, E., Muench, M., Spensky, C., Redini, N., Machiry, A., Fratantonio, Y., Balzarotti, D., Francillon, A., Choe, Y.R., Kruegel, C., et al., 2019. Toward the analysis of embedded firmware through automated re-hosting. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019), pp. 135–150.
Haller, I., Slowinska, A., Neugschwandtner, M., Bos, H., 2013. Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In: USENIX Security Symposium, pp. 49–64.
Hennessy, J.L., Patterson, D.A., 2011. Computer Architecture: A Quantitative Approach. Elsevier.
IDA Pro– Hex Rays. https://www.hex-rays.com/products/ida/. 2020.
ld.so(8) linux manual page. https://man7.org/linux/man-pages/man8/ld.so.8.html.
Lemieux, C., Sen, K., 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, pp. 475–485.
Liaw, J.,. QEMU Binary Translation. https://www.slideshare.net/RampantJeff/qemu-binary-translation.
Keystone Engine: Next Generation Assembler Framework. https://www.blackhat.com/us-16/briefings.html-keystone-engine-next-generation-assembler-framework.
Nagy, S., Nguyen-Tuong, A., Hiser, J.D., Davidson, J.W., Hicks, M., 2021. Breaking through binaries: compiler-quality instrumentation for better binary-only fuzzing. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 1683–1700.
Nie, S., Liu, L., Du, Y., 2017. Free-fall: hacking tesla from wireless to can bus. Briefing, Black Hat USA.
Pro, I.,. Survey Shows Linux the Top Operating System for Internet of Things Devices. https://www.linux.com/news/survey-shows-linux-top-operating-system-internet-things-devices-0.
preeny. https://github.com/zardus/preeny.
QEMU version 4.2.50 User Documentation: 1.1 Features. https://www.qemu.org/docs/master/qemu-doc.html-intro_005ffeatures.
Quynh, N. A.,. Capstone Engine. https://github.com/aquynh/capstone.
QEMU: a generic and open source machine emulator and virtualizer. https://www.qemu.org/.
Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H., 2017. VUzzer: application-aware evolutionary fuzzing. In: Proceedings of the Network and Distributed System Security Symposium (NDSS).
Sen, K., Marinov, D., Agha, G., 2005. Cute: a concolic unit testing engine for c. In: ACM SIGSOFT Software Engineering Notes, Vol. 30. ACM, pp. 263–272.
Song, D., Hetzelt, F., Das, D., Spensky, C., Na, Y., Volckaert, S., Vigna, G., Kruegel, C., Seifert, J.-P., Franz, M., 2019. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. NDSS.
Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2016. Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS, Vol. 16, pp. 1–16.
Wang, J., Chen, B., Wei, L., Liu, Y., 2017. Skyfire: data-driven seed generation for fuzzing. In: Security and Privacy (SP), 2017 IEEE Symposium on. IEEE, pp. 579–594.
Wang, T., Wei, T., Gu, G., Zou, W., 2010. Taintscope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Security and privacy (SP), 2010 IEEE symposium on. IEEE, pp. 497–512.
Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T., 2018. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium (USENIX Security 18), pp. 745–761.
Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D., et al., 2014. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. NDSS.
Zalewski, M., a. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl/.
Zalewski, M., b. ReadMe for American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/README.txt.
Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., Sun, L., 2019. Firm-afl: high-throughput greybox fuzzing of IoT firmware via augmented process emulation. In: 28th {USENIX} Security Symposium ({USENIX} Security 19), pp. 1099–1114.

Xuechao Du received his B.Eng. on computer science from Xian Jiaotong University, Xian, China, in 2016. He is currently a Ph.D. candidate in the college of Computer Science, Zhejiang University, China. His research interests include mobile security, Internet of Things security and software security based on program analysis.

Andong Chene received his B.Eng. on computer science from Shandong University, China, in 2019. He is a Ph.D. student in the college of Computer Science, Zhejiang University, China. His research interests include Internet of Things security and system security.

Boyuan He is a postdoctoral research associate at Department of Electrical Engineering and Computer Science, Northwestern University. He earned his a Ph.D. in computer science from Zhejiang University. His research interests lie in cybersecurity with the special focus on: logic vulnerability detection, Android app security, blockchain security, IoT device security, malware detection and forensic analysis.

Hao Chen is a full professor in the Department of Computer Science at the University of California, Davis. He received his Ph.D. at the Computer Science Division at the University of California, Berkeley, and his BS and MS from Southeast University. His research interests are computer security and machine learning.

Fan Zhang received his Ph.D. degree from the Department of Computer Science and Engineering, University of Connecticut, USA. He is currently an associate professor in School of Cyber Science and Technology, College of Computer Science and Technology, Zhejiang University, and affiliated with Alibaba-Zhejiang University Joint Research Institute of Frontier Technologise. His research interests include hardware security, system security, cryptography, and computer architecture.

Yan Chen received his Ph.D. in Computer Science from University of California at Berkeley in 2003 and after that he joined Northwestern University USA where he became a Full Professor in 2014. His research interests are in security and measurement for networking systems. Based on Google Scholar, his papers have been cited over 14,000 times, and the h-index of his publications is 56. He is a Fellow of IEEE.