AFL-IoT: Fuzzing on Linux-based IoT Device with Binary Level Instrumentation

Anonymous Submission

Abstract—**With the growing prevalence of Internet of Things (IoT), security and privacy issues on IoT devices have considerably increased. Due to limited resources, security analysis on IoT devices faces great challenges. On one hand, even though there are several existing works that provide automated solutions to identifying vulnerabilities on IoT devices, they are either inaccurate or not scalable. On the other hand, although sophisticated security analysis technique like fuzzing test has been studied for decades and has proved itself to be very effective in finding software bugs, none of existing fuzzing approaches could work with IoT devices.**

In this paper, we present AFL-IoT, a lightweight and easy deployable fuzzing framework for Linux-based IoT devices. Using binary level instrumentation, AFL-IoT could execute the instrumented binary program directly on IoT devices during the fuzzing process. We evaluated AFL-IoT on both benchmarks and real world IoT devices. In total, AFL-IoT identified 157 unique crashes in 10 binary programs with only 10% performance overhead on average, showing its good efficiency and effectiveness to find software bugs in binary programs on Linux-based IoT devices.

I. Introduction

Although it has been decades since the concept of Internet of Things (IoT) was first presented [1], only in recent years have we witnessed an outbreak of its application in our daily life. Various types of Commercial-Off-The-Shelf (COTS) IoT devices emerged rapidly, such as IP cameras, home routers, printers, smart TVs, lamps, fridges and air-conditioners. These devices extend the Internet connectivity beyond traditional devices (e.g. computers and smartphones) to any object we may use. Thus, it greatly expands the capability of the Internet and provides convenience for everyone.

However, the widespread application of IoT devices also brings new challenges for security and privacy. Interconnection of these devices that were originally isolated from each other significantly expands the attacking surface and thus makes IoT devices more vulnerable to cyber-attacks. For example, privilege escalation vulnerabilities have been found in smart lock and can be used by attackers to break the authentication [2]. Even worse, security researchers [3] have demonstrated that it is possible to penetrate into a car and gain remote control of Engine Control Units (ECU) simply with the help a phishing WiFi hotspot. Without physical access, both attacks were considered impossible in the past. Moreover, compared with traditional devices like smartphone, IoT devices often have relatively limited hardware resources and thus usually lack security mechanisms to prevent various attacks due to performance concern. That makes IoT devices even more exploitable.

To mitigate aforementioned security issues, many existing works attempted to find software vulnerabilities and flaws in IoT devices. Costin et al. [4] leveraged static analysis techniques to automatically acquire and analyze a large number of firmwares. They used specific patterns to search for existing security flaws in these firmwares. Although this approach is lightweight and scalable, it suffers from the inherited limitation of static analysis: accuracy.

Zaddach et al. [5] however took a dynamic approach to conduct security analysis on IoT devices by monitoring the firmwares execution on a hybrid system combining both emulator and real IoT hardware. The hybrid system automatically forwards all I/O accesses from the emulator to the real device and therefore precisely emulates the execution of the firmware. While this approach may be accurate, it still suffers from efficiency and scalability issues. Chen et al. [6] designed a dynamic analysis system for Linux-based IoT device's firmware. It takes an emulator-based approach to identify software vulnerabilities by performing large scale automated black-box testing with web exploits. Unfortunately, emulating various types of IoT devices is a non-trivial task. Their approach failed to resolve many dependencies such as a specific version of Linux kernel or certain kind of kernel modules before the OS can actually boot. Such failures may cause a large number of kernel panics during the emulation, which decreases the effectiveness of their approach.

To overcome the limitations of existing work, we take a different approach by using fuzzing in the security analysis of Linux-based IoT devices. Fuzzing is an effective technique to find software bugs. For example, AFL [7] is a well-known coverage-based gray-box fuzzer (or fuzzer), which mutates the input to trigger bugs in the target program. AFL has identified a large number of security vulnerabilities in many popular open source projects. However, although fuzzing has been studied in the field of software security for years, even today we still rarely see its application in IoT devices.

Compared with its application on x86 platforms, fuzzing on IoT devices poses the following major challenges:

1) **No source code.** Commercial-off-the-shelf (COTS) IoT devices are usually installed with many proprietary software. Generally, source code and documentation of such software are unavailable. All existing solutions for binary program fuzzing require runtime emulation, which is non-trivial on IoT devices.
2) **Limited resources.** In comparison with x86 platform, IoT devices often have limited resource in terms of CPU, memory and storage. So lightweight analysis is required on IoT devices. Trade-off between accuracy and efficiency must be taken into consideration in system design and implementation.
3) **Network daemon programs.** Compared with most x86 platform devices, network daemon programs are more common on IoT devices, and sometimes even outnumber

command line programs. These daemon programs take input from a network interface rather than a file or the standard input. Since traditional fuzzers can mutate only files or the standard input, we must design mechanisms to allow them to interact with daemon programs.

Despite of many existing works [8]–[21] focusing on research of various fuzzing tools, all of them fail to address these challenges. Consequently, unlike all these works, our goal is to propose a novel approach to adapt existing fuzzer to work with binary programs on IoT devices for efficient and comprehensive security analysis.

To this end, in this paper we design and implement AFL-IoT, a lightweight and easy deployable coverage-based fuzzing framework for Linux-based IoT devices. AFL-IoT extends AFL by implementing binary level instrumentation, so it can support native execution of the target binary program and therefore significantly reduces the performance overhead. In this way, AFL-IoT makes it possible to directly fuzz binary program on IoT devices instead of expensive runtime emulation and therefore addresses the first challenge.

Considering the limited resources on IoT devices, our solution consists of two phases for each target binary program in fuzzing: 1) Instrumentation phase. 2) Fuzzing phase. As shown in Figure 1, AFL-IoT first completes the instrumentation phase on a powerful Linux server and then deploys the instrumented program with the fuzzer on the IoT device. Our two phase design significantly eases the burden of IoT devices and thus addresses the second challenge.

Besides, our design also provides solutions to fuzzing on network daemon programs. Specifically, AFL-IoT can forward all input from a fuzzer to a specific network interface which the target daemon program listens on. Such forwarding is achieved through a series of hook operations on Linux socket APIs. In this way, AFL-IoT avoids expensive network operations and thus makes it more efficient for fuzzing on network daemon programs, which definitely addresses the third challenge.

To our best knowledge, we are the first to provide a practical fuzzing solution to binary programs on Linux-based IoT devices. We evaluate AFL-IoT on both benchmarks and real-world devices. Results shows that AFL-IoT achieves nearly the same performance on binary program in comparison with programs that AFL compiles from the source code and only has 10% performance overhead on average (See Section V-A2). In contrast, the performance overhead of the emulation-based approach that originally implemented in AFL ranges from 100% to 400% [22], which is obviously too expensive for most IoT devices.

Overall, AFL-IoT successfully identify 157 unique crashes in 10 binary programs on real-world Linux-based IoT devices in our evaluation (See Section V-B), showing that the approach that we apply in AFL-IoT is both efficient and effective to find software bugs in binary programs on Linux-based IoT devices.

The rest of this paper is organized as follow. Section II provides relevant background of AFL. Section III and Section IV depict the system design and implementation of AFL-IoT, respectively. Section V evaluates AFL-IoT on both benchmarks and real-world IoT devices. Section VI discuss the limitation of our work. Section VII introduces related works and Section VIII conclude the paper.

II. Background

A. American Fuzzy Lop (AFL)

American Fuzzing Lop (AFL) [7] is a famous coverage-based grey-box fuzzer which has been widely used to find software bugs in both academia and industry. Although its approach seems to be "naive' which simply makes mutations based on the origin input (seed) and keeps track of the execution of target program by using lightweight instrumentation, it has been proved to be very efficient and effective. AFL has gain a good reputation by identifying a large number of unknown bugs [7] in many popular opensource software projects. Next, we will go through some technique details to show basic ideas behind its design.

1) **Branch Coverage:** As a coverage-based gray-box fuzzer, AFL monitors the execution of the target program during the fuzzing process. Specifically, for each run of the target program, AFL maintains a 64KB map $shared\_mem[key]$ to keep track of the branch coverage information. It first generates a unique ID ($cur\_location$) for each basic block during the compilation and then instruments a small piece of code at each branch point to manipulate the map. If a certain basic block is hit during the execution, the instrumented code in this basic block increases the counter in $shared\_mem[cur\_location \oplus prev\_location]$ by 1 where $prev\_location$ is the shifted ID of the basic block that hit before the branch point. To distinguish the execution order between two basic blocks, the instrumented code right shifts $cur\_location$ by 1 bit before assigns it to $pre\_location$ ($pre\_location = cur\_location >> 1$). In this way, the entire execution path is recorded during each run of the target program and therefore AFL is capable of identifying new behaviors or internal states by comparing the executions paths that triggered by different inputs. All mutated inputs that can trigger new internal states within the target program are added to the input queue for further mutation in next runs, otherwise they are discarded. With the help of coverage-based guided fuzzing that described above, AFL is capable of exploring as many internal states of the target programs as possible, which significant increases the chance of finding bugs.

2) **Target Instrumentation:** The coverage measurement in AFL is implemented by instrumenting the target programs. Generally, such instrumentation is done during compilation and therefore requires source code of the target program. AFL provides two instrumentation approaches: assembly-level instrumentation and compiler-level instrumentation. The main difference is that the compiler-level instrumentation is performed on LLVM intermediate representation (IR) rather than manually instrumenting assembly code. Such LLVM-based instrumentation has several advantages: 1. It benefits from many optimization features that provided by the compiler so it is more efficient; 2. It is CPU-independent so it can be applied to fuzz on non-x86 platforms. Specifically, AFL provides wrappers for different compilers which can be used to replace the original compiler in the building script of the target program, so the instrumentation is done during the building process. However, both instrumentation approaches leverage on compilers it is necessary for AFL to rebuild the target program from the source.

In addition, AFL also has the capability to fuzz on binary program without source code. This feature is very useful when fuzzing many proprietary software and is implemented by leveraging on QEMU [23], an emulator that allows runtime instrumentation. While this emulator-based approach is feasible when fuzzing on binary programs, it still suffers from a major limitation. Emulator-based solutions are usually expensive because the emulator itself consumes large amount of resources. Besides, the emulator-based runtime instrumentation also has performance cost, which can significantly slow down the execution speed of the target program by two to five times [22]. Such performance overhead is probably unacceptable in some cases especially when fuzzing large binary programs.

III. Design

In this section, we present our system design of AFL-IoT and discuss technical details we apply in AFL-IoT.

A. Approach Overview

We design and implement AFL-IoT, a fast, lightweight and easy deployable coverage-based fuzzing framework for binary programs on Linux-based IoT device. The primary goal of AFL-IoT is to provide practical solution for security researchers to automatically analyze software on IoT devices where source code is usually unavailable.

Generally, there are several different approaches to implement coverage-based fuzzing on binary programs. The first approach that comes mind is to use virtual machine emulation. As mentioned in Section II, AFL leverages QEMU to hook instructions at each branch point in order to track execution of the target binary program. In this way, it achieves runtime instrumentation without modifying the target binary program. While this approach can provide the most accurate branch coverage information, it is too expensive to apply in IoT devices.

The next approach we can easily think of is assembly-level instrumentation which leverage on a disassembler as well as an assembler. Compared with the original binary program, assembly-level instrumentation only produces a small runtime overhead and all data references and function pointer access can be easily resolved by assembler after the instrumentation. However, this approach requires very high accuracy for both disassembler and assembler, which could be very challenging, especially for large complex binary programs.

Our Approach: Binary-level instrumentation. Considering the infeasibility of the above two approaches, we choose binary-level instrumentation in our design. That means to statically instrument the target binary program with binary code that tracks coverage information during the fuzzing process. In our approach, instrumented code is injected at each branch point to track the execution of each basic block. So, we first identify all basic blocks in the binary program, then inject binary code at the beginning of each basic block. In this way, our approach maintains all the code and data structures in the original program and therefore avoid breaking its code logic.

Generally, our approach has the following advantages: 1) It is lightweight and achieves almost the equivalent efficiency with source-code-level instrumentation (See Section V-A2). 2) It is compatible with many IoT devices because it fuzzes the target program on the original device and therefore does not require migration work such as environment configuration and dependence installation. 3) The accuracy of instrumentation will not affect the correctness of the target program execution. It will only affect the coverage information that are collected during the execution of the target program.

The system overview of AFL-IoT is illustrated in Figure 1. Specifically, there are two phases in the workflow of AFL-IoT: **instrumentation phase** and **fuzzing phase**.

In the instrumentation phase, we develop an elf-patcher and it first takes a binary program as an input, and then identifies and locates all of its basic blocks by disassembling and analyzing the binary code. Next, a small code stub for recording branch coverage is automatically instrumented into the beginning of each basic block that the elf-patcher identified. Finally, a code stub for initialization and communication with the fuzzer is automatically registered as an initialization entry of the target program. To guarantee the efficiency, all steps in the instrumentation phase are performed outside the target IoT devices.

In the fuzzing phase, both the instrumented program and the fuzzer are deployed on the original device before an instrumentation-guided coverage-based fuzzing is performed and all the fuzzing results are recorded. Besides, IoT devices usually contains many network daemon software (e.g. a web server), so we also design and implement an input redirection framework for fuzzing on network daemons.

Overall, AFL-IoT provides two major features: binary-level instrumentation and network input redirection. We will go through the technique details of these features in following subsections.

B. Basic Block Identification

The basic idea of instrumentation-guided coverage-based fuzzing is to improve execution coverage by monitoring each execution path of the target program for different inputs. To this end, each basic block of the target binary program needs to be identified with good accuracy before instrumentation. A basic block is a straight-line code sequence with no branches in except the entry and no branches out except at the exit [24]. Once the execution of the target program enters a specific basic block, all code inside this basic block must be sequentially executed.

As illustrated in Figure 1 AFL-IoT first leverages a static analyzer to perform reverse engineer on the target binary

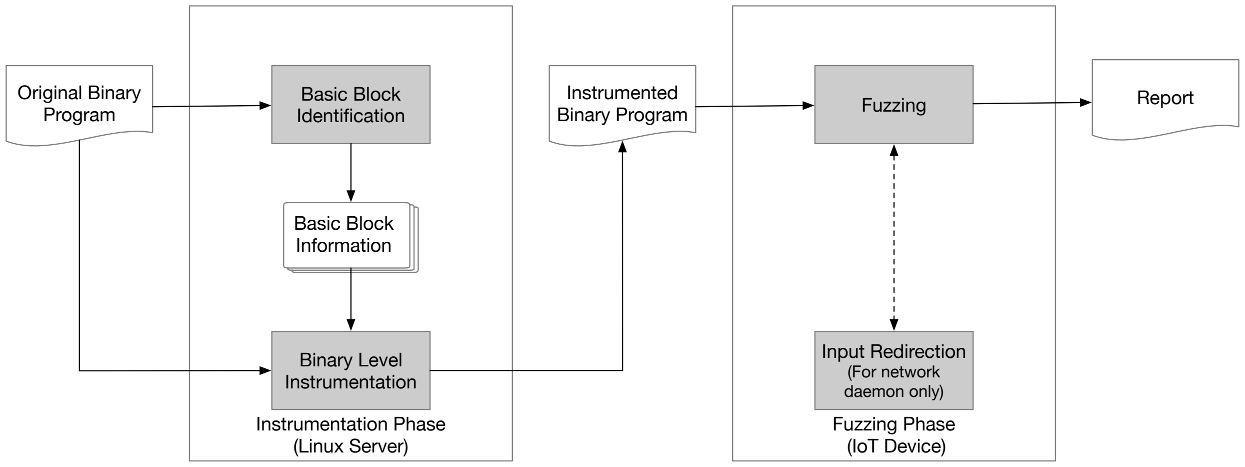

Original Binary Program
Basic Block Identification
Basic Block Information
Binary Level Instrumentation
Instrumentation Phase (Linux Server)
Instrumented Binary Program
Fuzzing
Input Redirection (For network daemon only)
Fuzzing Phase (IoT Device)
Report

Fig. 1: Overview of AFL-IoT.

program including generating a hierarchical call graph and extracting a list of functions. Then for each function, AFL-IoT checks if it can be triggered from entry point of the program by performing a backward search on the call graph. Such check eliminates those unreachable basic blocks from the results and therefore improves the efficiency of AFL-IoT. Next, AFL-IoT marks all reachable functions, and for each function it again leverages the static analyzer to search for all basic blocks within this function. Finally, AFL-IoT merges the searching result, eliminates duplicated basic blocks and logs all the information (e.g. type of instruction set, base address of the basic block, size of the basic block, etc.) of each basic block for later use.

C. Binary-level Instrumentation

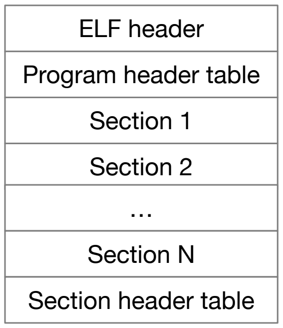


Fig. 2: Executable and Linkable Format (ELF) File Structure.

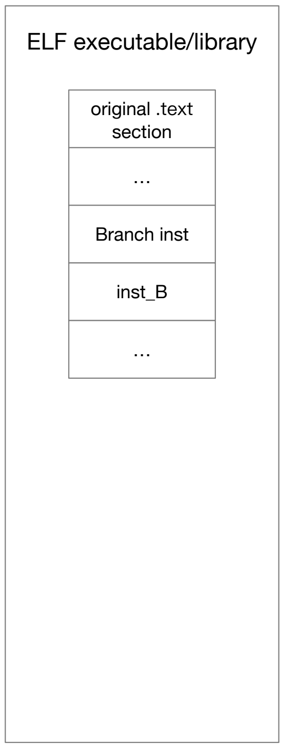
AFL-IoT targets at Executable and Linkable Format (ELF) [25] binary program which is used in most Unix-like operating system. The binary-level Instrumentation that perform by AFL-IoT is pure static and therefore only takes the target program and its basic block information as a input.

The structure of a typical ELF file is described in Figure 2. $ELF\ Header$ contains the basic information of the file, such as instruction set and architecture. Besides, $ELF\ Header$ also stores the position and size for both $Program\ Header\ Table$ and $Section\ Header\ Table$. $Program\ Header\ Table$ defines run-time information for the binary program. For example, PT_LOAD entries specific the mapping between sections and memory segments, $PT\_DYNAMIC$ entry specifics the position and size for $.dynamic$ section. $Section\ Header\ Table$ specifies position and size for each section within the ELF file. Such information is useful during the linking process. $Sections$ are segments where code and data are actually stored in an ELF file. There are various types of sections that defined in an ELF file, such as $.text$ section, $.data$ section, etc. Generally, $Program\ Header\ Table$ is not necessary for ELFs during linking while $Section\ Header\ Table$ is omittable for ELFs during run-time. However, most of compilers tend to keep both tables at the same time.

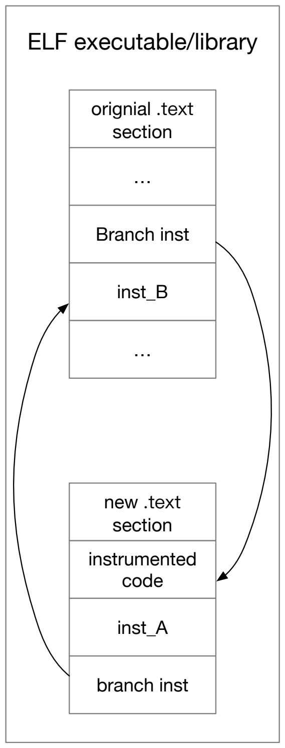
Our basic idea of instrumentation is to add code stub into each basic block without breaking the code logic of the original program. However, in a compiled ELF binary program, there is no space for the instrumented code in the original sections because the addresses for all the instructions and data have already been calculated by the compiler and moving such instructions or data at binary-level will likely disrupt the execution of the original program. Consequently, our solution makes extra space for instrumented code by adding new sections at the end of the ELF binary program.

As shown in Figure 3, for each basic block, we replace its first instruction $inst\_A$ with a branch instruction that redirects the program counter (PC) to a newly added section where our instrumented code is executed. Then, the replaced instruction $inst\_A$ is moved after the instrumented code. Finally, another branch instruction is added at the end of the new section to redirect PC back to the next instruction $inst\_B$ in the original section. Generally, our solution performs the following three steps for binary instrumentation:

1) Adding new sections for instrumented code and data and updating $Section\ header\ table$ for these new sections.
2) Adding new PT_LOAD entry in $Program\ header\ table$ to make sure all newly added data is loaded into the memory during execution.



(a) Original ELF file



(b) Instrumented ELF file

Fig. 3: Binary-level Instrumentation.

3) Replacing the original instructions and adding instrumented instructions and data.

1) **Adding New Sections:** The approach we take to add new sections in ELF binary program is quite straightforward. For new data section, we first calculate the length of the inserted data and determines the address and size of the new section. Then we add these sections directly at the end of the last section in the original program. Meanwhile, for each new data section, we will add a new entry in $Section\ header\ table$ to specify its address, offset and size. Moreover, since we expand the size of $Section\ header\ table$ which therefore increases the size of the ELF file, the ELF header also needs to be modified accordingly.

However, things are getting a little bit complicated when adding new code sections. Our code instrumentation relies on the assembler to generate new instructions. Without the size information of the newly added code section, the assembler has no idea of the section layout and relative offset from the original instructions to the inserted code stub, therefore the assembler cannot generate correct instructions for the new stub. However, the size of the new code section depends on the length of the instrumented instructions, which can only be determined by the assembler. In other words, there is a deadlock.

To solve this deadlock, our solution is to take two assemble passes. In the first pass, we assume the address for the original instructions with which the assembler is able to calculate the length of the instrumented instructions. In this way, the size of the new code section is determined and can be used in the second pass for code instrumentation.

Section	Description
.interp	Absolute path of the loader
.note.	Additional information about the ELF
.hash	Symbol hash table
.gnu.hash	GNU style symbol hash table
.dynsym	Symbol table
.dynstr	Strings used by .dynsym

TABLE I: Types of sections that may appears after $Program\ header\ table$ in an ELF binary program.

2) **Updating Program Header Table:** Sections are static information in an ELF binary program. Therefore the instrumented data and instructions will not be loaded into memory by simply adding new sections to the original ELF program. We need to add PT_LOAD entries for all new sections to specify the access permission for each segment of data and the memory address where it should be loaded to.

Unlike $Section\ header\ table$ which is located at the end of the ELF file (shown in Figure 2), expanding $Program\ header\ table$ will inevitably erase sections that comes after it. To avoid the such conflict, we come up with two solutions: 1. Moving $Program\ header\ table$ behind all sections. 2. Moving those sections that come after $Program\ header\ table$ backwards. Unfortunately, in practice the first solution is infeasible because Linux kernel assumes that $Program\ header\ table$ comes immediately after ELF header. In other words, it will result in a failure when running the instrumented program if we moved the $Program\ header\ table$ elsewhere. We therefore adopt the second solution. We first analyze a large number of ELF binaries that we collected from dozens of IoT device firmwares. The statistics show that only certain types of section (Table I) appears immediately after $Program\ header\ table$. All these types of sections have their location information stored in $Program\ header\ table$ or *.dynamic* section, so they can be relocated by simply modifying the pointers accordingly.

After moving all these sections, we can easily expand the $Program\ header\ table$ by adding new entries without any conflict. For each type of the sections that we added or moved, we add a corresponding PT_LOAD entry to $Program\ header\ table$. Besides, newly added sections are granted with one of the three access permission: $READ\_ONLY$, $READ\_WRITE$ and $READ\_EXECUTE$, which is reflected in their PT_LOAD entries.

3) **Wrapping Original Instruction:** As shown in Figure 3, our code instrumentation first replaces the first instruction $inst\_A$ in each basic block of the target program with a branch instruction and thus redirect the execution to our newly added .text section. Instrumented code, the replaced $inst\_A$ and another branch instruction are then sequentially inserted into the new section.

Binary level instrumentation could be very tricky for CISC architecture (e.g. x86), where instructions usually have variable lengths. So replacing one instruction with another may overwrite the subsequent instructions. Fortunately, most IoT devices are built upon RISC architecture (e.g. ARM, MIPS, etc.) with fixed instruction length. As a result, $inst\_A$ can be easily replaced with a branch instruction without touching the instruction that comes after it.

One major challenge to apply our code instrumentation is that we need to guarantee the successful execution of the original $inst\_A$ (See Figure 3) in the new location. To this end, our instrumented code maintains the values in all registers or memory addresses that are used in the original program. Specifically, our instrumented code first backs up all registers before using them and recovers those values at the end. For memory addresses, our instrumented code only writes to unused addresses below the current stack pointer to avoid potential conflicts.

Besides, some instructions use the value of program counter (PC). Since the value of PC varies at different position in a program, moving these PC-related instructions to new locations (e.g. $inst\_A$ in Figure 3) may disrupt the executions of the origin program and thus result in errors. To eliminate such effect caused by PC, in our approach we first divide all instructions into the following five types based on their different relationship with PC. Then for each type, we provide a different wrap solution to the original instruction.

- **T1: PC-independent instruction.** These are instructions that are completely independent from the value of PC. In other words, they are not related to any computation or memory access based on the value of PC.
- **T2: PC-dependent branch instruction.** Some branch instructions (e.g. b in ARM instruction set) depend on the value of PC to determine the destination address. An offset which is relative to PC can be used as an operand to indicate the destination address.
- **T3: PC read-only instruction.** These are the instructions that take the value of PC as one of their source operands, and then either assign it to another register or load a value from the memory with a relative offset to current PC.
- **T4: PC read-only instruction with stack operation.** These instructions take the value of PC as their source operand which works just like T3 except they also involve in stack operation.
- **T5: PC read-write instruction.** These instructions involve in both read and write operations that are related to PC.

Wrap Solution. For each type of instruction (T1 to T5), our approach provides a wrap solution (S1 to S5) to eliminate the effect caused by different PC values at new positions in the target program.

- **S1.** For PC-independent instruction (T1), it will not affect their execution by moving it to a new location. As a result, no specific wrap solution is needed other than simple branch back.
- **S2.** For PC-dependent branch instruction (T2), although its destination address is determined based on the value of PC, the destination address can be easily obtained by disassembling this branch instruction in the original program. In this way, the branch instruction becomes PC-independent and therefore does not require any specific wrap solution either.
- **S3.** For PC read-only instruction (T3), we first find a unused register as $pivot$ and save its original value to stack. Secondly we get the value of PC at the original position of the target instruction (e.g. $Inst\_A$ in Figure 3(a)) by using a disassembler and save it to $pivot$. Then we replace each occurrence of PC register with $pivot$ in the target instruction at new position (e.g. $Inst\_A$ in Figure 3(b)). In the end, the value of $pivot$ register is recovered from stack after the execution of the target instruction at new position.
- **S4.** Compared with T3 instruction, T4 instructions involves with both the value of PC and stack operations. Thus wrap solutions in S3 may cause conflict in this situation. Empirically, T4 instructions usually push a number of registers (including PC register) to stack. So in addition to S3, we also need to coordinate the position on stack for PC, $pivot$ and other registers. The basic idea behind this wrap solution is to avoid conflict in stack memory allocation when pushing $pivot$ and PC. Our wrap solution (S4) takes the following steps: 1. Set an unused register as $pivot$. 2. Push $pivot$ to stack for backup and load the original value of PC to $pivot$. 3. Push $pivot$ to stack again with target PC value on top the last push. 4. Recover $pivot$ from stack. 6. Push other registers.
- **S5.** For PC read-write instruction (T5), simply using S3 or S4 could lead to problems because T5 instruction sets new value to PC after execution and therefore redirects the execution of the program to somewhere else. Such redirection does not give us a chance to recover $pivot$ from stack, thus disrupts the execution of the program. To resolve these problems, we need to guarantee the recovery of $pivot$ the same time when redirect happens. Our warp solution (S5) takes the following steps: 1. Set an unused register as $pivot$ and push $pivot$ to stack for backup. 2. Load the original value of PC to $pivot$. 3. Replace each occurrence of PC register with $pivot$ in the target instruction at new position. The destination address of redirection RA will be written to $pivot$ after the execution of the target instruction. 4. Push $pivot$ to stack. 5. Pop stack twice to recover $pivot$ first and then load RA to PC for redirection.

With the five wrap solutions that listed above, our approach for code instrumentation can be successfully applied in various kinds of target binary programs on many Linux-based IoT devices. Examples that demonstrate how to implement each wrap solution are given in Section IV-D.

4) **Library Instrumentation:** In addition to ELF executable, shared library is another common format of Linux-based programs. Many software implements its core features in shared libraries, so it is necessary for AFL-IoT to provide binary-level instrumentation support for shared libraries. Otherwise, our coverage-based fuzzing will not be able to track execution paths inside shared library program, which significantly reduces its efficiency and effectiveness.

Code instrumentation on shared libraries is similar to that on ELF executables. However, there are still some challenges. For example, shared libraries are required to be compiled with position-independent code (PIC) support [26] to achieve dynamic loading, which is usually not the case for ELF executables. With PIC support enabled, the base address of a shared library can only be determined after the library being loaded into the memory and address relocation needs to be done once the base address is determined. Such relocation process requires careful consideration during the code instrumentation.

Next, we describe how to address these two challenges in our approach.

Fuzzing Initialization. A coverage-based fuzzer usually requires to instrument a piece of code at the beginning of the target program for initialization purpose. In our approach, we leverage on $.init\_array$ section in an ELF file to execute initialization code in the target program. $.init\_array$ is an array where each element specifies a function to be executed at the beginning of the execution. As a result, by instrumenting new elements into $.init\_array$, we can successfully execute fuzzing initialization code in a shared library.

Relocation. Code instrumentation on a target binary that compiled with PIC support inevitably needs additional work of relocation. Specifically, we need to take relocation into consideration in the following three situations: 1. Moving or modifying any original data sections. 2. Instrumenting code into the original code sections. 3 Instrumenting code or data into the newly added sections.

As to the first situation, since $.init\_array$ is the only original data section that we move or modify in our code instrumentation, we only need to perform relocation for every new element that we add into $.init\_array$ section.

For the second situation, if our code instrumentation replaces original instructions with new instructions, we simply need to delete the previous relocation and performs new relocation for the newly added instructions.

For the last situation, we first record all the symbols that occur during instrumentation, find locations where these symbols are referenced by comparing each piece of instrumented code and then perform relocation for each symbol in the order of occurrence.

After resolving the relocation issue, the instrumented shared library can be used to track path information during our coverage-based fuzzing process, which obviously increases the efficiency and accuracy of AFL-IoT.

D. Fuzzing on Network Daemons

Most of fuzzers target at programs that take input from a local file (e.g. a parser program) or standard I/O streams (e.g. a command-line program). In coverage-based fuzzing, this input is constantly changed by the fuzzer according to the coverage information that it collected during the execution of the target program. However, things are completely different on IoT devices. Since IoT devices are ideally designed to fulfil the connectivity of different "things", network servers or daemons have become the most dominant type of software on IoT devices. Instead of reading input from file or standard I/O streams, network daemons listen on certain ports and interact with clients from other devices. So the major challenge to fuzz on network daemons is to find an approach for efficient input redirection.

The first approach that can be easily think of is to set up a transparent proxy, which acts as a client for the target network daemon. Whenever the fuzzer starts, instrumented code stub inside the target daemon is first triggered and the proxy is then notified to send new input that generated by fuzzer. As one its advantages, this approach does not require any modification to the code logic of the original network daemon, so it is easy to implement and deploy. However, network operation is too expensive to be involved in fuzzing. For example, the latency for a network daemon to bind on a certain port is usually several milliseconds on Linux-based IoT devices, which is apparently unacceptable in fuzzing. Consequently, a proxy-based approach is infeasible.

Since the major overhead comes from network operation and such operations are usually manipulated by socket APIs, our approach performs an efficient input direction by hooking socket APIs to map input from standard I/O stream to network sockets. In this way, expensive network operations at OS level are successfully avoided and the efficiency of input redirection is tremendously improved. In our approach, we first specify a port number $target\_port$ before fuzzing starts and therefore all the operations of the target daemon program on this port will be mapped to standard I/O stream accordingly. The reason to specify $target\_port$ is to avoid conflict that caused by the target daemon program listening on multiple ports simultaneously. Secondly, we hook $bind()$ API to check if it is going to bind on $target\_port$. If so, an unnamed pair of connected sockets are then created by $socketpair()$ API to act as a full duplex pipe for input/output redirection. Specifically, in our approach we hook the following three types of socket APIs:

1) Basic socket APIs (e.g. $socket()$, $bind()$, $listen()$, $accept()$ and $close()$).
2) Socket APIs that specify file descriptors to listen on in user-space (e.g. $select()$ and $poll()$).
3) Socket APIs that register file descriptors to listen on in kernel-space (e.g. $epoll\_create()$, $epoll\_ctl()$ and $epoll\_wait()$).

Next, we illustrate the detailed hook operations when each of the following socket APIs is called in the target daemon program.

- **socket()** The $socket()$ API is used to create socket file descriptor. For a newly created socket file descriptor, we have no idea if it is our target socket for input redirection until it is bind to a certain port. So here we only label this file descriptor for later use.
- **bind()** At this point, port number is already specified by parameters. If it matches $target\_port$, we mark its file descriptor as $bind\_fd$ and return success immediately. Otherwise, call the original $bind()$ API.
- **listen()** Return success immediately if the socket file descriptor matches $bind\_fd$. Otherwise, call the original $listen()$ API.
- **accept()** If the socket file descriptor does not matches $bind\_fd$, call the original $accept()$ API. Otherwise, create a socketpair (as a pipe) of the same type with $bind\_fd$ and mark the two ends of the pipe as A and B. Then create two threads to redirect from $stdin$ to B and B to $stdout$ respectively. Finally return A as $accept\_fd$.
- **close()** If the socket file descriptor does not matches $bind\_fd$, call the original $close()$ API. Otherwise, exit the target daemon program immediately and notify the fuzzer to start the next instance.

In general, hooking the basic socket APIs that listed above would be sufficient to fuzz on most network daemon programs except for those listening on a list of file descriptors asynchronously. To make our approach more complete, we also hook the following two APIs:

- **select()/poll()** $select()$/$poll()$ is used specify a list of socket file descriptors to listen on. As mentioned previously, we return success immediately without actually binding $bind\_fd$ to a certain port. So we will never get notification on $bind\_fd$ even if $bind\_fd$ is in the file descriptor list of $select()$/$poll()$ API. As a result, we simply remove $bind\_fd$ from the list and then call the original $select()$/$poll()$ API. Finally, we manually add event notification for $bind\_fd$ on return.
- **epoll** $epoll$ is another set of socket APIs that are used to listen on multiple file descriptors asynchronously. The major difference is that the file descriptor list that created by $epoll$ is maintained in kernel-space. We take similar operation as in $select()$/$poll()$ to remove $bind\_fd$ from the list before calling to original $epoll$ API. In addition, $epoll$ allows the program to attach a 64-bit long data to each file descriptor, so we need to record all the operations on $epoll$ to provide correct information for $bind\_fd$ on return.

IV. Implementation

In this section, we describe technical details as well as challenges to implement AFL-IoT.

A. Fuzzer integration

Recall that our goal is to perform efficient coverage-based guided fuzzing on binary programs in Linux-based IoT devices. To this end, our work is mainly focused on how fuzzer interacts with the target binary program instead of improving its fuzzing strategies. As a result, we adopted AFL directly into our fuzzing framework, leverage on instrumented code to track branch coverage information and make it fully compatible to work with AFL. This also guarantee that any improvement from AFL community will also benefits our system without any modifications. Beside, we compiled the AFL initialization code into a shared library, so it can be loaded into target program before the initialization phase of fuzzing, and then the fork-server and shared memory in AFL will be initialized by the code stub that we instrument into the target program via specific interfaces in the shared library.

B. Device Setup

As mentioned in previous sections, hardware resources are extremely limited on the most of IoT devices, where most vendors will deploy a streamlined runtime library such as $uClibc$ instead of standard C library $libc$, causing problems when fuzzing on these devices. AFL requires certain symbols in $libc$ to execute correctly. However, such symbols are sometimes missing on the IoT devices where $uClibc$ [27] is deployed. Our solution is to performs static compilation specifically for each device, so additional library dependencies are unnecessary.

Besides, the ELF loader that comes with some IoT devices does not enable TLS (Thread local storage)-related support. Therefore programs with TLS variables cannot be loaded on these devices. AFL requires TLS support to track the execution path of different thread in the target program. Fortunately, we notice that even though sometimes the ELF loader may not provide TLS support, however, the Linux kernel is compiled with TLS feature enabled. Thus we can easily fix the problem by just rebuilding a new version of ELF loader with TLS support enabled for the target IoT device and switch to this new loader when necessary.

In addition, most COTS IoT devices do not preserve debug interfaces for shell access, which is a basic requirement for fuzzing on device. Fortunately, such debug interfaces do exist in most devices for factory testing. For security reasons, these interfaces are usually blocked after passing all tests. So we use several tricks to activate these debug interface in order to enable shell access.

C. Basic Block Identification

Most binaries from IoT devices are stripped, which can be challenging for reverse engineering tools to completely identify the code that resides in the binary. But as we mentioned in previous sections, a failure in identifying certain basic blocks will only affect path resolution accuracy in AFL and does not cause any error in fuzzing. We test several existing opensource reverse engineering tools, such as Barf [28], Sibyl [29] and Miasm [30], but none of them could provide acceptable accuracy. So we end up leveraging on IDA Pro [31], a well-known commercial reverse engineering tool. It retrieves basic blocks information from the target binary program with good accuracy which meets the requirement of binary level instrumentation.

```
loc_1000: add r1, pc, r2
loc_1004: ...
loc_1008: ...
...
...
```

(a) Original Code

```
stmdb sp, {r3}
ldr r3, =loc_1008
add r1, r3, r2
ldmdb sp, {r3}
b loc_1004
```

(b) After Instrumentation

Fig. 4: Wrap Solution for T3 instructions (ARM).

```
loc_1000: push {r0, r2, r4, pc}
loc_1004: ...
loc_1008: ...
...
...
...
...
```

(a) Original Code

```
sub sp, sp, 4
stmdb sp, {r1}
ldr pivot, =loc_1008
stmia sp, {r1}
ldmdb sp, {r1}
push {r0, r2, r4}
b loc_1004
```

(b) After Instrumentation

Fig. 5: Wrap Solution for T4 instructions (ARM).

```
loc_1000: ldr pc, [pc, r3, lsl#2]
loc_1004: ...
loc_1008: ...
...
...
...
...
```

(a) Original Code

```
sub sp, sp, 4
stmdb sp, {r1}
add sp, sp, 4
ldr pivot, =loc_1008
ldr r1, [r1, r3, lsl#2]
stmdb sp, {r1}
ldmdb sp, {r1, pc}
```

(b) After Instrumentation

Fig. 6: Wrap Solution for T5 instructions (ARM).

D. Instruction Wrapping

As mentioned in Section III-C3, to guarantee successful execution of the original instruction (e.g. $inst\_A$ in Figure 3) at new position, we provide different wrap solutions for various types of instructions in binary level instrumentation. We successfully apply all these solutions (Section IV-D S1-S5) in AFL-IoT for ARM binary programs. Although our current implementation of AFL-IoT only targets at binary programs on ARM platform, these wrap solutions are general and can be ported to binaries on other platforms (e.g. MIPS) without much effort. Next, we give examples to demonstrate how we implement these wrap solutions, specifically S3, S4 and S5 for T3,T4 and T5 instructions. We omit T1 and T2 instruction here because their wrap solutions (S1 and S2) are relatively simple to understand.

For PC-read-only instruction (T3), it reads from PC but writes somewhere else. For example, $addr1, pc, r2$ is a typical T3 instruction on ARM platform. As we described in S3, first we need to find a unused pivot register, save its original value to stack, and store the value of PC. Here we choose $r3$ as a pivot register and the wrap solution for $addr1, pc, r2$ is illustrated in Figure 4.

For PC read-only instruction with stack operation (T4), it acts just like T3 instructions except involving stack operation. As a result, we need to coordinate the position stack for PC, pivot register and other registers. For example, $pushr0, r2, r4, pc$ is a typical T4 instruction on ARM platform. If we choose $r1$ as a pivot register, then the wrap solution for $pushr0, r2, r4, pc$ is illustrated in Figure 5.

For PC read-write instruction (T5), it involves in both read and write operations that are related to PC. Figure 6 demonstrates how we implement S5 for a typical T5 instruction $ldrpc, [pc, r3, lsl\#2]$ where $r1$ is chosen as the pivot register.

We implemented our binary level instrumentation framework for ELF binaries by leveraging on keystone-engine [32] as an assembler and capstone-engine [33] as a disassembler. Both of them support a variety of architectures, making it possible for our system to easily adapted to binary programs from other architectures. Overall, our binary level instrumentation framework is implemented in python with 5300 lines of code.

E. Input Redirection

For network daemon programs, we implement an efficient input redirection mechanism between a fuzzer and a target network daemon programs. Specifically, we implement a shared library called $libsockets$ in which we rewrite all of the common Linux socket APIs (See Section III-D). With dynamic linking, we successfully hook all these socket APIs in the execution of the network daemon programs using $libsockets$ and performs checks on parameters to determine whether to intercept or forward network traffic. In total, the implementation of input redirection takes about 300 lines of python code and 3000 lines of C code.

V. Evaluation

In this section, we evaluate AFL-IoT on both benchmarks and real-world IoT devices in terms of its correctness, performance and effectiveness.

A. Evaluation on Benchmarks

As mentioned in previous sections, AFL-IoT allows coverage-based fuzzers (e.g. AFL) to run on Linux-based IoT devices by implementing binary level instrumentation. Although our approach does not involve any modification to fuzzing strategy of the original fuzzer, we leverage a well-known fuzzer benchmark: LAVA-M [34] to evaluate its correctness and performance.

1) Correctness: To verify the correctness of our approach, we need to run AFL-IoT on a dataset with ground-truth to see if it can correctly fuzz a program and find bugs. LAVA-M is a widely-used fuzzer benchmark that automatically injects a large number of bugs into four GNU core utilities programs: $uniq$, $base64$, who and $md5sum$. We selected the $uniq$ program to compare AFL and AFL-IoT. Since AFL cannot run on real device, we ran this experiment on Ubuntu 16.04 in the QEMU-ARM emulator, where each host is configured with 2 cores and 4GB memory.

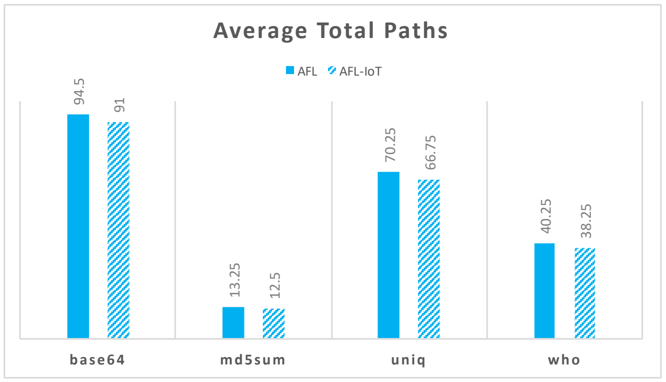
We ran two experiments on $uniq$ program in parallel, one using AFL and the other using AFL-IoT. To mitigate the impact of randomization during fuzzing, we repeated each experiment six times. In the experiment with AFL, we first compiled the $uniq$ program from the source code with afl-clang-fast and then ran it with AFL on a single core for 10 hours. AFL performs a LLVM level instrumentation during compilation. In comparison, our AFL-IoT experiment performed binary level instrumentation on a pre-compiled version of $uniq$ program. We compiled it using the same source code and parameters of afl-clang-fast as in the AFL experiment. Similarly, the experiment lasted for 10 hours on a single core.

We used several metrics provided by AFL to measure the fuzzing process. First, the number of unique crashes indicates the identified bugs. Second, the branch coverage indicates the proportion of branches that were executed in the fuzzing process. Third, the number of total path reflects the number of unique execution paths triggered by fuzzer. Table II shows that the AFL-IoT and AFL experiments have similar results, which demonstrates the correctness of AFL-IoT.

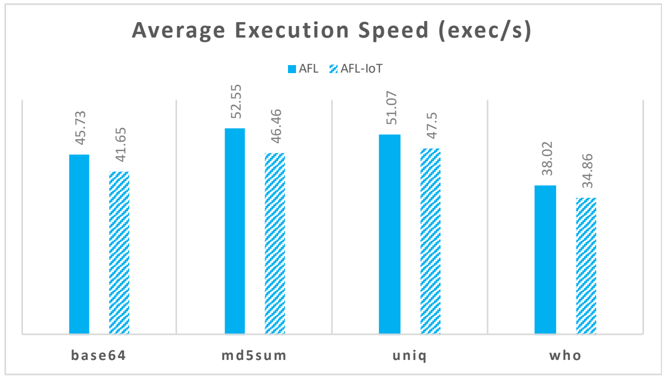
2) Performance: In addition to correctness, performance is also vital for fuzzing. Since AFL-IoT performs binary level instrumentation on target programs, it lacks source code level information, making instrumented code optimization even impossible. As a result, programs that instrumented at the binary level inevitably suffer from performance overhead in comparison with those with source code instrumentation. To evaluate AFL-IoT, we compared the fuzzing test performance between source code level instrumentation by AFL and binary level instrumentation by AFL-IoT. We again chose the LAVA-M dataset. This time we tested AFL and AFL-IoT on all the four programs in LAVA-M with the same experiment environment as in our previous evaluation: dual-core CPU with 4GB memory under QEMU-ARM emulator. Similarly, for each target program, we built two versions: one compiled from source code with afl-clang and then run with AFL, another compiled with clang using the same parameters and then run with AFL-IoT after binary level instrumentation. To mitigate the influence of randomness, we ran each version of the four

Tool	# of Run	# of Unique Crash	Branch Coverage	# of Total Path	Execution Time (H)
AFL	1	6	29.82%	75	10
	2	0	29.82%	80	10
	3	4	29.82%	77	10
	4	0	29.82%	78	10
	5	4	29.82%	83	10
	6	4	29.82%	71	10
AFL-IoT	1	6	29.64%	79	10
	2	1	29.64%	76	10
	3	1	29.64%	74	10
	4	4	29.64%	82	10
	5	3	29.64%	76	10
	6	1	29.64%	78	10

TABLE II: Fuzzing Test Result on $uniq$ in Lava-M Dataset with AFL and AFL-IoT.



(a) Average Total Paths



(b) Average Execution Speed

Fig. 7: Performance Comparison Between AFL and AFL-IoT.

program four times. The test of each running instance lasted for one hour on a single core.

We chose the number of total paths and execution speed as indicators of performance. Figure 7 compares the performance between AFL and AFL-IoT. For all the four target programs, there is a 5% average decrease for AFL-IoT with instrumented programs in comparison with AFL with compiled programs in terms of number of total paths. The main reason of this decrease is that the instrumented programs ran slower than the compiled ones, resulting in a decrease in the number of instances that the fuzzer can execute in the same time. This affects the number of input that the fuzzer can mutate and therefore reduces the number of total paths.

Among all four programs in LAVA-M dataset, the instrumented versions have an execution speed drop of about 10% on average compared with the compiled versions. This performance overhead can be attributed to the following three reasons:

1) As AFL-IoT performs instrumentation at binary level, it does not have enough context information, which makes it impossible to optimize the instrumented code stub. Therefore, AFL-IoT must save all used registers every time when it enters the instrumented code stub. In contrast, AFL's source-based instrumentation can take full advantage of the compiler's optimization capabilities to avoid saving context frequently. Besides, it can also use idle registers to store data such as pointers globally, thus yielding a faster execution of instrumented code stubs.
2) As mentioned in the design and implementation sections, we need to wrap the original instruction at the target address during the binary level instrumentation to ensure that the overwritten instruction can be executed as expected in the instrumented code segment. This inevitably introduces pivot-related operations together with expensive memory access instructions, which will slow down the execution of the target program.
3) Finally, to guarantee the integrity of the original program, we place the instrumented code and data at the end of the entire program. When loaded into the memory for execution, the program needs to jump to the instrumented code frequently to record the execution path. Since the original code segment of the program is far away from the instrumented code segment in memory, they cannot be loaded into the CPU cache at the same time. This frequent back and forth jump can cause a large number of instruction cache misses, which further adds additional overhead to the execution of the program.

Overall, compared with AFL, AFL-IoT achieves the similar code coverage performance, only about 5% decrease in average number of total path and about 10% decrease in average execution speed on binary programs. With slight performance overhead, AFL-IoT is fully capable of fuzzing binary programs on most Linux-based IoT devices, where resources are very limited.

B. Evaluation on Real-world IoT devices

The ultimate goal of AFL-IoT is to fuzz binary programs in COTS Linux-based IoT devices. Since we have evaluated AFL-IoT on benchmarks for its correctness and performance, next we can evaluate it on real-world IoT devices. Our evaluation selects 4 popular ARM Linux-based home routers from three well-known device manufacturer: ASUS AC1700, Xiaomi R1D/R3D and Netgear R7000.

We upgraded all the four devices to the latest version of firmware and got shell access on each device using the debugging interfaces. Next, we selected dozens of command-line programs and daemon programs from each device. We determined how to interact with these programs and where to feed input to them by analyzing their behaviors. Since these programs are designed to run on these devices, they can be executed directly without further environment configuration or dependency resolution. For programs that read from the standard input or from a file, we performed binary level code instrumentation with with AFL-IoT directly. For those designed to read input from network (e.g. daemon programs), we additionally applied our approach described in Section III-D to redirect the input to specific network interfaces and ports. Overall, our evaluation performed binary level instrumentation and fuzzing on over 100 binary programs on the four devices. For each fuzzing instance, we used a single-core CPU and ran within 24 hours.

Table III shows the crashes that AFL-IoT found in 10 different binary programs. The first seven programs are archive tools included in Busybox, a popular Linux utility toolbox for IoT devices. AFL-IoT found as many as 30 unique crashes in these programs. Further analysis shows that most of the bugs have been fixed in the latest version of Busybox. Notably, one crash from lzcat and two crashes of unlzma were fixed just a few months before our evaluation, which demonstrates AFL-IoT's ability to find real software bugs. In addition to the bugs in Busybox, AFL-IoT also found 12 unique crashes in the independent archive program unzip. The ninth program in Table III is a third-party version of sar (System Activity Reporter) [36], in which AFL-IoT found 14 unique crashes. The last one, plugincenter, is a daemon program from Xiaomi R1D smart router. Although daemon programs generally needs to do a series of initialization when starting, which could slow down the execution speed, AFL-IoT still found 22 unique crashes in it.

In summary, AFL-IoT is a fast, lightweight and easily deployable coverage-based fuzzing framework for binary programs on Linux-based IoT device. The major advantage of AFL-IoT over other fuzzers is that it directly instruments the target program at the binary level, which makes it possible to fuzz various types of IoT devices with slight performance overhead. Our evaluation shows that AFL-IoT is accurate and effective in finding bugs in binary programs on real world Linux-based IoT devices, and it only has about 10% runtime overhead in comparison with AFL.

No.	Binary Program	Program Type	Device	Execution Time (H)	Branch Coverage	Total Paths	Execution Speed (exec/s)	Unique Crashes
1	bunzip2[1]	command-line	Xiaomi R1D	14	2.27%	203	82	11
2	bzcat[1]	command-line	Xiaomi R1D	12	2.27%	202	111	13
3	gunzip[1]	command-line	ASUS AC1700	14	3.19%	204	35	20
4	gzip[1]	command-line	Netgear R7000	12	3.38%	255	124	30
5	lzcat[1]	command-line	Xiaomi R1D	14	0.96%	78	76	1
6	unlzma[1]	command-line	Xiaomi R1D	12	1.00%	79	94	4
7	zcat[1]	command-line	Netgear R7000	12	3.23%	224	103	30
8	unzip	command-line	ASUS AC1700	12	45.57%	626	278	12
9	tsar	command-line	Xiaomi R1D	14	22.82%	45	22	14
10	plugincenter	network daemon	Xiaomi R1D	14	11.32%	262	16	22

[1] These programs are common Linux utilities that are integrated into a single executable called Busybox [35]. Since it combines many different programs in one executable, the branch coverage for each of them is relatively low.

TABLE III: Fuzzing Test Result in Real-world COTS Linux-based IoT Devices with AFL-IoT.

VI. Limitation

Although AFL-IoT has been proved as an lightweight fuzzing tool for Linux-based IoT devices with good efficiency and accuracy, we still acknowledge the following limitations:

- **Accuracy of binary level instrumentation.** As described in previous sections, AFL-IoT first leverage IDA Pro [31] to identify all basic blocks in the target binary program and then for each basic block it performs binary level instrumentation. As a reverse engineer tool, IDA Pro usually achieves good accuracy. However, it is still possible for it to miss a small number of basic blocks in the target program, which can eventually affect the accuracy of binary level instrumentation in our approach. Empirically, IDA Pro can miss up to 5% of basic blocks. Please note that such inaccuracy will only result in loss of branch coverage information in fuzzing process and will not affect the correctness of program execution.
- **Efficiency of fuzzing on network daemon.** For network daemon programs, AFL-IoT performs input redirection by hooking socket APIs. Although such feature has eliminated major performance overhead that caused by network operations, daemon programs may still be relatively slow in fuzzing. Our observation shows daemon programs usually run 1 to 2 times slower than command-line programs.

VII. Related Works

A. IoT Device Security

With the increasing popularity of IoT device, many existing works focus on the security issues on IoT devices. In recent years, large number of software vulnerabilities on various types of IoT devices have been identified and many attacks against IoT devices have been revealed. Oluwafemi et al. [37] demonstrate the possibility of attacks against compact fluorescent lamps by compromising Internet-enabled home automation systems, showing that non-networked devices might be connected to networked devices and hence can be attacked by remote adversaries. Similarly, Notra et al. [38] evaluate the several household devices in terms of security and privacy and shows that they can be easily compromised by remote attackers due to security flaws. Ronen et al. [39] study the multiple smart light device and show attackers can even use smart light devices to remotely exhilarate data from a highly secure building with LIFI communication system. The leaked data can be read by attackers from a distance over 100 meters. Ho et al. [40] examine the security of several home smart locks that can be controlled remotely by user or manufacturer. Their work reveals multiple security flaws in design and implementation of smart locks which can be used by attackers to access private information about users and gain unauthorized access. Sivaraman et al. [41] demonstrate a infiltration attack into home network from Internet via a malicious mobile app. Such attack takes advantage of vulnerability in home router to scan for potential vulnerable IoT devices inside home network, collect information and modify the firewall to allow the external entity to directly attack the IoT device. Moreover, Zhang et al. [42] reveals the fact that many home assist devices with speech recognition engine are vulnerable to inaudible attack. With malicious hidden voice commands, attackers are able to remotely manipulate many popular home assist device without users notice. Such unauthorized access poses great risk to users leading to privacy leakage or other security issues.

To mitigate the risk of various attacks against IoT devices, several works have made their effort to perform automated security analysis on IoT devices. Costin et al. [4] present the first large-scale analysis of firmware images with static analysis techniques. They build a system to collect a large number of firmware from various device vendors, and unpack all these firmware images into millions of files. Their analysis reveals a large number of vulnerabilities and some of them are even shared by many different devices. Although their work found a great number of vulnerabilities, it still suffers from a limitation of accuracy due to their shallow static analysis approach. To improve accuracy, Zaddach et al. [5] propose a hybrid approach by leveraging both emulator and physical device to perform dynamic analysis. In their approach, firmware instructions are executed inside the emulator but all I/O operations are channeled to the physical devices. While this approach enables full emulation of the target device, execution switch between emulator and physical device is expensive and thus is not scalable. Besides, their implementation is specific to certain types of device, so it is not easy to generalize their system to many different devices. Chen et al. [6] present a dynamic security analysis framework for Linux-based IoT devices. First, it collect a large number of firmwares from various vendors. Then, it automatically unpack and configure the firmware to run in an emulator. Finally it perform a large-scale black-box testing with web exploits on the emulator. However, it is impossible to perform comprehensive security analysis with web exploits. Besides, it is a non-trivial task to resolve software dependence and hardware configuration issues in order to prevent kernel panic during the emulation. However, their approach fails to address these issues well, making it unusable with many firmwares.

Overall, all existing work in the field focus on attack, mitigation or simple security analysis on IoT devices. None of the them provides capability of comprehensive security analysis with fuzzing techniques in order to find previously unknown bugs on IoT devices. To the best of our knowledge, we are the first to provide practical fuzzing solution to IoT devices.

B. Fuzzing

There is a large number of related works in the research field of fuzzing. To improve seed selection for mutation-based fuzzers, AFLFast [9] uses Markov Chain to identify "low-frequency paths" and focuses most of effort on these paths because it is more likely to trigger bugs when fuzzing on these "low-frequency paths". VUzzer [8] leverages on control-flow graph to select input. To improve coverage, several works implement dynamic symbolic executions, such as DART [13], SAGE [12], KLEE [10], Driller [11], CUTE [14] and SYM-FUZZ [15]. Besides, since dynamic taint analysis can find dependencies between the input and the program logic, it is now widely used in many fuzzers, such as BuzzFuzz [17], Taintscope [18], Dowser [16] and Angora [43]. In addition, there are also several works that adopt learning techniques to generate input in fuzzing (e.g. Skyfire [21], Learn&Fuzz [20] and GLADE [19]).

As mentioned in previous sections, IoT devices usually have very limited resources, lack source code and require addition configuration for fuzzing. None of these existing work addresses all these challenges. However, unlike all these works, our goal is to adapt existing coverage-based fuzzer to work with binary programs on IoT devices. Specifically, we design and implement AFL-IoT, a lightweight and easy deployable fuzzing framework for binary programs on IoT devices.

VIII. Conclusion

In this paper, we present AFL-IoT, a lightweight and easy deployable coverage-based fuzzing framework for Linux-based IoT devices. To our best knowledge, AFL-IoT provides the first practical fuzzing solution for binary programs on IoT device. With binary level instrumentation, AFL-IoT supports native execution of the target program on Linux-based IoT devices and therefore significantly reduces the performance overhead. We evaluate AFL-IoT on both benchmarks and real-world IoT devices. Overall, AFL-IoT is able to identify 157 unique crashes in 10 binary programs on 4 devices and only has 10% performance overhead on average in comparison with AFL. Our evaluation shows that AFL-IoT is both efficient and effective to find software bugs in binary programs on Linux-based IoT devices.

References

[1] "Wikipedia: Internet of things." [Online]. Available: https://en.wikipedia.org/wiki/Internet_of_things

[2] J. Max, "Backdooring the frontdoor hacking a perfectly secure smart lock." 2016. [Online]. Available: https://media.defcon.org/DEFCON24/DEFCON24presentations/DEFCON-24-Jmaxxz-Backdooring-the-Frontdoor.pdf

[3] S. Nie, L. Liu, and Y. Du, "Free-fall: Hacking tesla from wireless to can bus," *Briefing, Black Hat USA*, 2017.

[4] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis, "A large-scale analysis of the security of embedded firmwares." in *USENIX Security Symposium*, 2014, pp. 95–110.

[5] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares." in *NDSS*, 2014.

[6] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *NDSS*, 2016.

[7] "American fuzzy lop." [Online]. Available: http://lcamtuf.coredump.cx/afl/

[8] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[9] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, 2017.

[10] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[11] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[12] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, p. 20, 2012.

[13] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.

[14] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 263–272.

[15] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 725–741.

[16] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: a guided fuzzer to find buffer boundary violations." in *USENIX Security Symposium*, 2013, pp. 49–64.

[17] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 474–484.

[18] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Security and privacy (SP), 2010 IEEE symposium on*. IEEE, 2010, pp. 497–512.

[19] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM, 2017, pp. 95–110.

[20] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 50–59.

[21] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 579–594.

[22] "Readme for american fuzzy lop." [Online]. Available: http://lcamtuf.coredump.cx/afl/README.txt

[23] "Qemu: a generic and open source machine emulator and virtualizer." [Online]. Available: https://www.qemu.org/

[24] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[25] "Position-independent code." [Online]. Available: https://en.wikipedia.org/wiki/Position-independent_code

[26] "Executable and linkable format." [Online]. Available: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

[27] "uclibc: Embedded c library." [Online]. Available: https://www.uclibc.org/

[28] C. Heitman and I. Arce, "Barf: A multiplatform open source binary analysis and reverse engineering framework," in *XX Congreso Argentino de Ciencias de la Computación (Buenos Aires, 2014)*, 2014.

[29] "A miasm2 based function divination." [Online]. Available: https://github.com/cea-sec/Sibyl

[30] "Miasm: Reverse engineering framework in python." [Online]. Available: https://github.com/cea-sec/miasm

[31] "Ida pro." [Online]. Available: https://www.hex-rays.com/products/ida/index.shtml

[32] "Keystone engine: Next generation assembler framework." [Online]. Available: https://www.blackhat.com/us-16/briefings.html#keystone-engine-next-generation-assembler-framework

[33] "Capstone: Next generation disassembly framework." [Online]. Available: https://www.blackhat.com/us-14/briefings.html#capstone-next-generation-disassembly-framework

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

[35] "Busybox: The swiss army knife of embedded linux." [Online]. Available: https://busybox.net/

[36] "System activity report (sar)." [Online]. Available: https://en.wikipedia.org/wiki/Sar_(Unix)

[37] T. Oluwafemi, T. Kohno, S. Gupta, and S. Patel, "Experimental security analyses of non-networked compact fluorescent lamps: A case study of home automation security." in *LASER*, 2013, pp. 13–24.

[38] S. Notra, M. Siddiqi, H. H. Gharakheili, V. Sivaraman, and R. Boreli, "An experimental study of security and privacy risks with emerging household appliances," in *Communications and Network Security (CNS), 2014 IEEE Conference on*. IEEE, 2014, pp. 79–84.

[39] E. Ronen and A. Shamir, "Extended functionality attacks on iot devices: The case of smart lights," in *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 2016, pp. 3–12.

[40] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner, "Smart locks: Lessons for securing commodity internet of things devices," in *Proceedings of the 11th ACM on Asia conference on computer and communications security*. ACM, 2016, pp. 461–472.

[41] V. Sivaraman, D. Chan, D. Earl, and R. Boreli, "Smart-phones attacking smart-homes," in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, 2016, pp. 195–200.

[42] G. Zhang, C. Yan, X. Ji, T. Zhang, T. Zhang, and W. Xu, "Dolphinattack: Inaudible voice commands," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 103–117.

[43] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search." in *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2018.