An Efficient Greybox Fuzzing Scheme for Linux-based IoT Programs Through Binary Static Analysis

Yaowen Zheng[1,2], Zhanwei Song[2,*], Yuyan Sun[2], Kai Cheng[1,2], Hongsong Zhu[1,2], and Limin Sun[1,2]
[1] School of Cyber Security, University of Chinese Academy of Sciences, China
[2] Beijing Key Laboratory of IoT Information Security Technology,
Institute of Information Engineering, Chinese Academy of Sciences, China
{zhengyaowen, songzhanwei, sunyuyan, chengkai, zhuhongsong, sunlimin}@iie.ac.cn, *corresponding author

***Abstract*—With the rapid growth of Linux-based IoT devices such as network cameras and routers, the security becomes a concern and many attacks utilize vulnerabilities to compromise the devices. It is crucial for researchers to find vulnerabilities in IoT systems before attackers. Fuzzing is an effective vulnerability discovery technique for traditional desktop programs, but could not be directly applied to Linux-based IoT programs due to the special execution environment requirement. In our paper, we propose an efficient greybox fuzzing scheme for Linux-based IoT programs which consist of two phases: binary static analysis and IoT program greybox fuzzing. The binary static analysis is to help generate useful inputs for efficient fuzzing. The IoT program greybox fuzzing is to reinforce the IoT firmware kernel greybox fuzzer to support IoT programs. We implement a prototype system and the evaluation results indicate that our system could automatically find vulnerabilities in real-world Linux-based IoT programs efficiently.**

***Index Terms*—Fuzzing, Linux-based IoT program, vulnerabilities discovery, embedded security, static binary analysis.**

I. Introduction

Nowadays, IoT (Internet of Things) systems have been widely deployed and the number will exceed 20 billion by 2020 [1]. Among heterogeneous IoT devices, many devices such as network cameras and routers take Linux kernel as their operation system and just make some modification to it. For Linux-based IoT system, developers are more concerned about functionality implementation and system performance optimization and do not treat security design and implementation adequately. Meanwhile, Linux kernel is open source and widely used, so that attackers are familiar with it. As a result, Linux-based IoT systems are prone to attack if there are vulnerabilities existing in programs. For example, Mirai botnets utilize vulnerabilities to compromise network cameras which lead to extremely destructive damage in the whole cyberspace. Therefore, it is crucial for security researchers to find vulnerabilities in Linux-based IoT systems before attackers.

Fuzzing is an automated software testing technique, that provides random input to program and monitor the exception such as crashes [2]. It is an effective way to find software vulnerabilities. Greybox fuzzing [3] is the one of fuzzing techniques that collects information from program execution by lightweight instrumentation techniques to guide the generation of random input which makes fuzzing more efficient. American fuzzy lop (AFL) [4], one of the most popular greybox fuzzing tools, can collect code coverage in a lightweight way to guide fuzzing procedure, helpful in detecting vulnerabilities for lots of desktop programs efficiently.

Compared with desktop programs, Linux-based IoT programs have the following specific characteristics. First, source code and design documents are often proprietary, and we could only obtain binaries by extracting firmware [5]. Second, Linux-based IoT programs run in their specific environment supported by particular runtime configuration and I/O hardware. For the first discrepancy, current fuzzers (e.g., AFL) could support binary-only app instrumentation to achieve greybox fuzzing which makes it not a problem again. However, due to execution environment discrepancy and uncertainty, traditional fuzzing techniques could not be directly applied to Linux-based IoT programs.

Several recent works partially solved IoT fuzzing problems (including Linux-based) by either directly fuzzing to real devices (e.g., IoTFuzzer [6]) or through accurate IoT emulation (e.g., Avatar [7], Firmadyne [8] and TriforceAFL [9]). Among these IoT emulation schemes, Avatar proposed a hybrid emulation approach by combining software emulation with real hardware, while Firmadyne presented a full system emulation scheme. Furthermore, TriforceAFL integrated the AFL with full emulation approach to achieve full system fuzzing.

1) Challenges: Even adopting the state-of-the-art IoT emulation [7] [8] and fuzzing techniques [6] [9], efficient greybox fuzzing for Linux-based IoT programs is still not applicable due to the following reasons: First, the input generation mechanism is not efficient for IoT programs. Recent works still follow AFL input generation mechanism without changes which generates inputs by random seeds mutation. Since most IoT programs take structured data as input, most of generated inputs could be discarded by IoT programs at a very early stage. As a result, the fuzzing process could not execute deeper paths in IoT programs which reduces fuzzing efficiency. Although IoTFuzzer [6] improves the input generation mechanism due to the involvement of mobile app analysis, it is still

978-1-7281-1025-7/19/$31.00 ©2019 IEEE

impractical since most IoT systems do not have related apps.

Second, it is difficult to collect target IoT program execution information for greybox fuzzing. Although TriforceAFL implements greybox fuzzing with full system emulation, it only works successfully with kernel fuzzing tests. Unlike system calls execution, the target program would execute concurrently with other processes and may switch to other ones and back at any time. Thus, collecting accurate code coverage information and observing the program state in real-time is a hard task.

2) Our solutions: To solve these challenges, we propose our efficient greybox fuzzing scheme for Linux-based IoT programs through binary static analysis. The system contains two parts: binary static analysis and IoT program greybox fuzzing. Binary static analysis is to figure out keywords that could determine the different execution paths for IoT programs. We then take these keywords into input generation process to improve IoT program fuzzing efficiency. IoT program greybox fuzzing is to reinforce IoT firmware kernel greybox fuzzer (TriforceAFL) to implement accurate IoT program execution observation and code coverage collection which guarantees the correctness for fuzzing. We keep the TriforceAFL workflow unchanged, and perform modification to the current modules. To complete the reinforcement, we also present real-time monitors including process monitor, system call monitor and context analyzer to support high-level modules.

To evaluate our system, we implement our system based on IDA Pro [10] and TriforceAFL. We then collect six firmwares (seven programs) including cameras and routers for our evaluation and utilize Firmadyne to configure these firmwares (using Linux kernel 3.2.1). Next, we evaluate the effectiveness of our system by finding real-world vulnerabilities in IoT programs, and finally evaluate the efficiency of our system by comparing discovered crashes and paths with random fuzzing. Through our evaluation, we show that (1) the effectiveness of our system since it indeed finds vulnerabilities in IoT programs, and (2) the efficiency which means our system could find vulnerabilities more quickly than random input generation fuzzing.

3) Contributions: In summary, we make the following contributions in our paper.

- To the best of our knowledge, this paper is the first to propose an efficient greybox fuzzing scheme in full system emulation through binary static analysis for IoT programs. It guarantees the correctness of IoT program greybox fuzzing through reinforcement of IoT firmware kernel greybox fuzzer and solves the efficiency problem with the help of binary static analysis.
- We implement a prototype of system and perform evaluation on real-world IoT programs. The results show that our system could efficiently find vulnerabilities in Linux-based IoT programs.

II. Overview

In this section, we present our greybox fuzzing scheme for IoT programs. The whole system could be divided into two phases: binary static analysis and IoT program greybox fuzzing. In the binary static analysis phase, we extract readable strings in binary executable from IoT firmware and discover useful keywords used as part of seed to improve fuzzing efficiency. In the IoT program greybox fuzzing phase, we reinforce the IoT firmware kernel greybox fuzzer to support IoT programs greybox fuzzing.

A. Binary static analysis

In this subsection, we discuss how we obtain useful keywords used as part of IoT fuzzing seed. As shown in Figure 1, we first obtain candidate keywords for the target binary program from IoT firmware (step 1, 2), and for each candidate keyword, we then perform taint analysis (step 3, 4, 5, 6) to analyze how the program parses and consumes it, and as a result, determine if the string is a keyword that could improve the fuzzing code coverage if involving in input generation. Through our analysis, we could find the strings that determine the execution path. Here, we define this kind of strings as `fuzz keyword`. Therefore, we first introduce how we get candidate `fuzz keyword`s from IoT firmware. Then, we perform taint analysis on them to determine final `fuzz keyword`s.

1) Candidate keywords obtaining: Since many Linux-based IoT firmwares are available from network, we could get them and extract the target binary program (step 1) and then obtain candidate keywords from the target binary (step 2).

In the step 1, we obtain the target binary from Linux-based IoT firmware. Here, we use Binwalk [11], a fast firmware images extraction tools, used by start-of-art work [8] to analyze the firmware format and extract all files from Linux-based IoT firmwares. We then find the target binary through file name identification.

In the step 2, we extract all readable strings from the target binary as our original candidate keywords sets for fuzzing. In an ELF style binary executable, strings are stored in .data or .rodata section, and each string often ends with $\backslash 0$ character. We also split the long string into small parts if $\backslash r$, $\backslash n$, $\backslash t$ control characters are involved in the string. Meanwhile, we get memory address of each string since we need to conduct taint analysis for them in next steps.

2) Taint source and sink: Generally speaking, taint analysis technique contains three major elements including taint source, taint propagation, and taint sink. It is usually applied to analyze taint-style vulnerabilities and external attack problems, and would take the variables that store external inputs as the sources and sensitive operations (APIs) as sinks. However, in our taint analysis problem, we deal with internal constant strings rather than variables that store external input, and we define taint source and sink as below.

- Taint source: The taint source is the pointer variable that stores the memory address of readable strings. Generally, we can identify these variables through locating the memory loading instruction to these address.
- Taint sink: IoT programs take structured data as input, and normally execute in different paths depending on different comparison results with `fuzz keyword`. Normally,

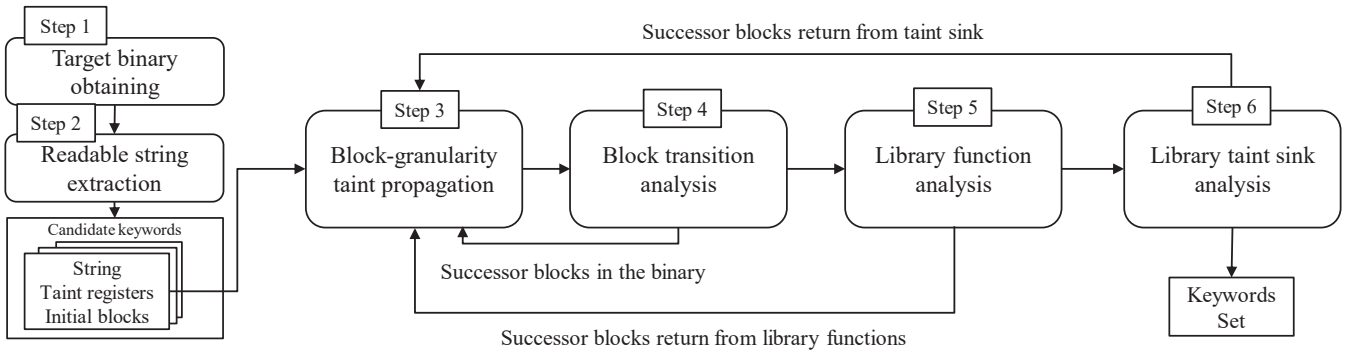

Fig. 1. Overview of binary static analysis

the binary executable compares the input with `fuzz keyword` using library function including memory and string comparison APIs. Thus, we define these library APIs as taint sinks shown in Table I.

3) Taint analysis procedure: Based on the definition of taint source and sink, we propose our taint analysis methodology illustrated in Figure 1. To make taint analysis more compatible, we break the taint analysis process into block granularity. We construct a queue to store blocks that are not analyzed. The initial block in the queue is the one that contains loading operation of the current analyzed string. For each block out from queue, we perform block-granularity taint analysis (step 3), and then perform block transition analysis (step 4) to identify successors if they are into library function or normal code blocks. For normal code blocks, we add them into the queue for further analysis. For library function block, we perform step 5 for library function taint analysis. Next, for library function that is defined as taint sink, we perform taint sink analysis (step 6) to determine if the taint analysis is completed. If the taint sink is not related to the analyzed string, we continue to get next block successors and add them into the queue. Obviously, we conduct block-granularity taint analysis in breadth-first-search mode. We describe each step in detail as below.

In the step 3, we perform block-granularity taint analysis. According to different analysis results after step 4, 5, 6, we repeatedly take this step and finally complete the whole taint analysis process. Initially, the input is the registers that are marked as tainted. Then, we propagate tainted variables instruction by instruction. When the taint analysis propagates to the end of block, we update the tainted registers.

In the step 4, we perform block transition analysis after step 3. We first get address and function property of the successor blocks. If the next successor blocks are still in the same function, we perform taint analysis (step 3) for these successor blocks. If next successor blocks are not in the same function, that means the block calls a sub function at the end of it. If the sub function is a normal one that code implementation is still in the binary executable, we also perform taint analysis (step 3) for it. Otherwise, the sub function is a library API, we then perform library function analysis (step 5).

In the step 5, we perform library function analysis. If the analyzed library function is a taint sink, we perform taint sink analysis in step 6. Otherwise, we get the taint flow summary from Table I to complete taint propagation analysis and then perform taint analysis (step 3) for blocks after the taint sink. we discuss our taint flow summary generation procedure in the next subsection.

In the step 6, we get the related registers of the taint sink from the last column of Table I and current tainting registers from step 3. if one of current tainting registers is contained in the related registers, we take the current analysis candidate string as a `fuzz keyword`. Otherwise, the analyzed string is not related to this sink, and further analysis will continue.

4) Taint flow rules generation for library function: We have two ways to perform library function taint propagation. One is to conduct inter-procedural taint analysis like normal function handling. Another is to generate taint flow summary through intra-procedural data-flow analysis of function implementation for one time and utilize the taint flow summary for every subsequent taint analysis. Both methods need to find its implementation in library binary from LIB directory and conduct data flow analysis on it. We choose the latter method since it only performs data flow analysis once for each library function.

Therefore, we start to collect library functions widely used by IoT programs and then filter out functions that are related to our analysis and then perform data flow analysis to generate taint flow summary for them.

- Library function collection and filtering: In Linux-based IoT firmware, programs frequently use `uClibc` [12] which is a small C standard library intended for embedded Linux-based operating systems. We download `uClibc` source code from [13] and get all library functions implementation in different C source files. Since we perform taint analysis on library functions that are related to strings, we filter out the string-related functions and mark other functions as non-related. Through our investigation, we find that most of string-related functions are implemented in `libc` root directory and distributed

in `stdio`, `stdlib`, `string` sub-directory. Here, we list representative functions from the library in Table I.

- Taint flow summary generation: We perform data flow analysis on these string-related functions and generate taint flow summary. Here, we define $a0$, $a1$, $a2$, $a3$ as first four arguments for the library function and $v0$ as the return value. we describe each item of summary like $A \longleftarrow B$ shown in Table I. It means if the string that B points is tainted or untainted, the string that A points is also marked as tainted or untainted correspondingly. Taint flow summary of each library function may contain several items. Moreover, we show the related registers of taint sink in the last column of Table I.

B. IoT program greybox fuzzing

In this section, we reinforce full emulation greybox fuzzer (TriforceAFL) to support IoT program greybox fuzzing. TriforceAFL is a modified version of AFL that integrates QEMU emulator [14] which supports kernel fuzzing for full-system emulation. It invokes system calls with random input and collects code coverage information to guide input generation for fuzzing. However, since it lacks real-time monitor and accurate code coverage collection for specific process during execution, fuzzing for IoT programs is not supported. In our design, we keep the workflow of TriforceAFL unchanged, and make modification to existing components based on our newly introduced monitor components to support IoT program greybox fuzzing. To the rest, we first describe the workflow of TriforceAFL and its key components, and then describe our modification to existing components. At last, we present our new components to achieve real-time monitor for IoT programs.

1) Workflow of TriforceAFL: TriforceAFL first starts IoT emulation including booting operating system. At the same time it would invoke a fuzz drive. When system initialization is complete, the fuzz drive would start AFL fork server which forks a child process and executes subsequent operation in it. In the child process, drive would get a new input from fuzz engine, feed it into target system call and exit to parent process when the system call is completed. There are several key components in the system.

a) Fuzz engine: The client side component of TriforceAFL. The fuzz engine follows the input generation strategies of AFL entirely. Initially, the seed pool only contains the initial seeds specified by users, and the fuzz engine would choose one of them, perform mutation on it to generate input, and finally feed the input into the target system call. With the fuzzing iterations going on, the mutated input that causes program to reach new code coverage would be stored as a new seed in the seed pool.

b) Fork mode: As discussed in the workflow, the fuzz drive would start AFL fork server to make subsequent execution in the fork of virtual machine. That means when a test case is completed, the fuzzing process rolls back to the fork point for next iteration. It avoids fuzzing process restarting from the beginning of system initialization which improves the fuzzing throughput. However, the fuzz drive is suitable only for system call fuzzing since it invokes system call right after fork server starting. For IoT program greybox fuzzing, since the target program starts up with specific configurations which we could not identify automatically. Therefore, we need propose new fork mode mechanism to monitor process starting in system emulation and fork at the specific point.

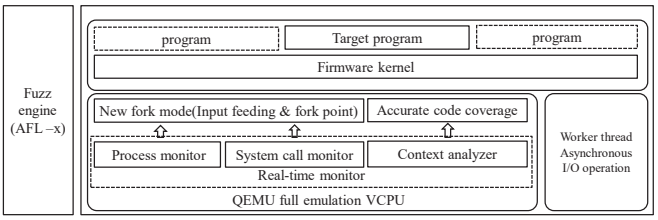

Fig. 2. Framework of IoT program greybox fuzzing

In addition, the fork mechanism is also different from AFL. AFL directly invokes fork since only one single-threaded program running in QEMU user mode. In full system emulation, multi-threaded programs run concurrently in the system. In order to fork correctly, TriforceAFL first exits CPU loop thread and then performs fork in the IO thread, and finally restores the CPU state and restarts CPU loop thread to continue execution.

c) Code coverage: To collect the code coverage information, TriforceAFL instruments the branch transition in QEMU system mode. Specifically, in every code block execution of system call, the instrumentation encodes the current program counter (PC) with the last block execution PC, and then stores the value in the bitmap shared with fuzz engine. After the system call execution is completed, the fuzz engine would compare the bitmap with previous accumulative bitmap, and determine if the new mutated input trigger the program to reach a new code coverage. If so, this input will be taken as a new seed and stored in the seed pool. In addition, TriforceAFL has solved greybox fuzzing problems in full system emulation such as how to collect code coverage accurately when the system interrupt happens and code blocks are re-executed.

2) IoT program greybox Fuzzing with TriforceAFL: In this subsection, we will describe the modification to several key components in TriforceAFL in order to adapt IoT program gerybox fuzzing. The framework of IoT program greybox fuzzing is shown in Figure 2.

a) Fuzz engine: The fuzz engine in TriforceAFL performs random mutation on seed to generate input for target program which is not suitable for IoT service programs anymore. Here, we use keywords filtered out from static binary analysis to participate in input generation. Specifically, since TriforceAFL (same with AFL) provides a way to seed the target program with a dictionary of keywords by enabling -x mode, we just provide our `fuzz keywords` directly and do not change any mutation strategies of TriforceAFL.

b) Fork mode: As we discussed above, TriforceAFL uses the fuzz driver to start fork mode before starting the targets (system call). For IoT service programs that receive input from network, since the network-related system call is

TABLE I
LIBRARY FUNCTIONS PROPERTIES

Function types	Function name	Taint flow summary	Taint Sink
Printable function	puts, fputs, printf, sprintf, snprintf, vsnprintf, fprintf	N/A	False
File-system related function	open, fopen, system, rename, remove, unlink ,rewind, stat, fstat, lstat	N/A	False
Memory manipulation operation	memcpy, memncpy, memmove	$a0 \longleftarrow a1$	False
	memcmp	N/A	True $(a0, a1)$
String manipulation operation	strdup, strndup, strdupa, strndupa, strchr, strrchr, strstr, strrstr	$v0 \longleftarrow a0$ $v0 \longleftarrow a1$	False
	strcpy, strncpy, strcat	$a0 \longleftarrow a1$	False
	strcmp, strncmp, strcasecmp, strncasecmp	N/A	True $(a0, a1)$
	strlen	N/A	False
Others	getenv	$v0 \longleftarrow a0$	False
	getopt	$v0 \longleftarrow a2$	False

time consuming, the program starts from very beginning in every iteration would extremely reduce the fuzzing throughput. Therefore, we specify the fork point at the network-related system call, and complete input feeding by hooking these system call. The implementation requires a real-time monitor of system calls execution which we will discuss in the next subsection.

In addition, we start a new worker thread in child process after fork to handle the asynchronous I/O operations. Otherwise, the program execution would hang at some system calls without asynchronous feedback.

c) Accurate code coverage: TriforceAFL used to fuzz Linux kernel and the code coverage is among the kernel virtual address space. However, user programs running in the virtual machine share the same virtual address space and would switch constantly. In order to get the code coverage only for the target program, we need to determine if current execution is in the target process before recording and encoding the PC. Therefore, we present a current context analyzer to support it.

3) Real-time monitor for IoT program: In order to achieve the new fork mode and accurate code coverage mechanism we mentioned before, we design a real-time monitor for IoT programs. The monitor have three functionalities shown in Figure 2: (1) process monitor which observes when the program starts and terminates, (2) system call monitor (especially network-related system calls) for new fork mode mechanism, and (3) context analyzer used to analyze current process context to support accurate code coverage collection. All the monitors are implemented by instrumenting the QEMU system mode.

a) Process monitor: We traverse task structure in firmware Linux kernel to monitor the process starting and termination. Specifically, when TLB-missed happens, we get the `task_struct` from Linux kernel structure to find if a new task is involved and old task is ended. If there is new task, we obtain task information including name, process identifier (PID), etc. if the name equals to target program name, the monitor notify the fuzzer that target program starts to execute. Similarly, we could know when the target program terminates according to the missing record.

b) System call monitor: For system call execution monitor, we instrument right after the system call exception generation to get system call information such as system call number and mark it as the beginning point of the system call. We also record PC and stack pointer before invoking the system call. Then, we instrument at the end of each block to get PC and stack pointer and compare them with our previous records. If both values are identical, the execution of system call is completed and returns back to the target process. Specifically, we monitor the network-related system calls to complete new fork mode and input feeding.

c) Context analyzer: The operating system sets up page global directory (PGD) when a process is created. The PGD values are different among different processes, so it could be used to present a process. For systems in MIPS architecture, we get current PGD value from Linux kernel structure to figure out current process that occupies the CPU to execute. For systems in ARM architecture, we get current PGD value from the specified register of system control coprocessor (CP15).

III. IMPLEMENTATION AND EVALUATION

A. System implementation

We have implemented our system using python and C languages respectively for static binary analysis and IoT program greybox fuzzing.

In static binary analysis phase, we use IDA Pro [10] to reverse IoT programs from binaries to assemble languages and reconstruct control flow and data reference information. We then implement taint analysis process in python code with the help of IDAPython [15]. In IoT program fuzzing phase, we implement our system based on TriforceAFL [9].

B. Experiment setup

Since there are many supported firmwares from Firmadyne datasheet [16] that belong to D-Link, Trendnet brands and corresponding device types are either network cameras or routers, we choose six firmwares (camera or router) from these brands as our testing sets. We utilize Firmadyne [8] to configure these Linux-based IoT firmwares including creating disk images and inferring the networking configurations. For all these firmwares, most of functionalities are implemented in HTTP service. Thus, we take seven HTTP-related programs from these firmwares as our testing targets. The basic information of testing programs is shown in Table II.

TABLE II
TESTING PROGRAMS INFORMATION

ID	Program	Vendor	Model	Version	Device
1	hedwig.cgi	D-Link	DIR-815	1.01	Router
2	hnap	D-Link	DIR-817LW	1.00B05	Router
3	hnap	D-Link	DIR-850L	1.03	Router
4	httpd	D-Link	DAP-2695	1.11.RC044	Router
5	httpd	D-Link	DIR-825	2.02	Router
6	video.cgi	Trendnet	TV-IP110WN	1.2.2	Camera
7	network.cgi	Trendnet	TV-IP110WN	1.2.2	Camera

All the testing programs could be categories into HTTP programs and CGI programs. The HTTP program handles the initial HTTP request while the CGI program handles the specific request on the back end of the HTTP program.

To evaluate the effectiveness of our system, we first evaluate our system on these seven programs to see if it could find real-world vulnerabilities. More specifically, we first perform static analysis to each target program and generate `fuzz keyword` set for input generation. Then, we construct a normal HTTP or CGI request as the initial seed for each target program fuzzing. In the fuzzing phase, fuzzer would select `fuzz keyword`s to generate inputs for target program according to AFL input generation rules.

For `fuzz keyword` set generation, we take the following strategies to make our system perform better. For CGI programs, the network request is encoded in "name=value" format and stored in environment variables. we find out `fuzz keyword`s that are environment variable names, and add symbol "=" to their tails. Meanwhile, we remove environment variable names in `fuzz keyword`s whose corresponding values could not be modified such as `CONTENT_LENGTH`, `REQUEST_URI`, `REMOTE_ADDR`, `REMOTE_PORT`. For video.cgi and network.cgi, we do not find any environment variable names in `fuzz keyword`s, so we add `QUERY_STRING` to the set. For hedwig.cgi, we add symbol "=" after "uid" keyword. In addition, for all the programs, we add a long string at the end of `fuzz keyword` set to make it possible to trigger the buffer overflow vulnerabilities more quickly.

We then use AFL, one of the most widely used greybox fuzzing tools for common programs and TriforceAFL to test these programs as the comparison.

To evaluate the efficiency, we compare our system with trivial input generation methods such as random testing. For random testing, we skip the static analysis phase and directly fuzz on these programs. To make evaluation results more convincing, we run each fuzzing experiment five instances for 24 hours.

We conduct our fuzzing experiments on a server with 40-core Intel Xeon(R) E5-2687W(v3) 3.10GHz CPU and 62.8GB of RAM. The operating system is Ubuntu 16.04 LTS.

C. Effectiveness

By performing the fuzzing to seven programs shown in Table II, our system successfully found known and unknown vulnerabilities shown in Table III, which illustrates the effectiveness of our system.

TABLE III
DISCOVERED VULNERABILITIES

ID	Program	known (ID)	unknown (NUM)	Triforce -AFL	AFL
1	hedwig.cgi	EDB-ID-33863	0	×	✓
2	hnap	EDB-ID-38720	0	×	×
3	hnap	CVE-2017-3193	0	×	×
4	httpd	CVE-2016-1558	1	×	×
5	httpd	N/A	0	×	×
6	video.cgi	CVE-2018-19241	0	×	×
7	network.cgi	CVE-2018-19240	0	×	×

TABLE IV
STATIC ANALYSIS RESULT

ID	Program	Strings (Num)	Keywords (Num)	Portion	Overhead (s)
1	hedwig.cgi	371	113	30.46%	43.30
2	hnap(DIR-817LW)	649	158	24.35%	87.77
3	hnap(DIR-850L)	643	156	24.26%	80.98
4	httpd(DAP-2695)	769	137	17.82%	150.94
5	httpd(DIR-825)	3033	356	11.74%	1081.67
6	video.cgi	28	12	42.86%	3.54
7	network.cgi	67	36	53.73%	1.01

We found known vulnerabilities for most of programs except httpd program on DIR-825 D-Link router, and we show related CVE number or Exploit-ID for them in Table III. We also found one unknown vulnerability when testing the httpd program on DIR-2695 D-Link router. Specifically, the request header that starts with "GET /upload_ca" and ends with "_int HTTP/1.1" would trigger this unknown vulnerability.

In contrast, the evaluation results of TriforceAFL and AFL are shown in last two columns of Table III. Most programs could not run correctly in AFL due to the lack of files which are generated during system initialization. Although hedwig.cgi on DIR-815 router could execute correctly in AFL, the fuzzing of program environment variable is not supported by AFL. For TriforceAFL, due to lack of real-time monitor for IoT programs, it could not support fuzzing of IoT programs in full emulation.

D. Efficiency

We first evaluate the binary static analysis phase and then compare the crashes and paths found over time with random testing for all testing programs to evaluate the fuzzing efficiency.

The binary analysis result is shown in Table IV. The results shows that the binary static analysis could extract keywords for fuzzing from original strings at the portion from 11.74% to 53.73%.

We then evaluate the fuzzing efficiency for IoT programs. To prove the importance of binary static analysis, we perform the random testing (without `fuzz keyword` set) as the comparison. We show the crashes and paths found over time respectively in Figure 3 and Figure 4. For each plot, red lines represent the fuzzing results with `fuzz keyword` set, and

blue lines represent the random testing results. Since we ran every fuzzing experiment five instances, you could see at most 5 lines each color in one plot (some fuzzing instances did not find any crashes in 24 hours, so the results could not be seen in the plot). In addition, we do not show the testing results of httpd on DIR-825 router since no crash was found. From the results, we can see that in most cases, our system could find more paths than random testing or almost the same with it. Furthermore, in spite of large variation across different fuzzing instances, our system could find crashes more quickly than random fuzzing.

IV. Limitation

In the paper, we have implemented efficient greybox fuzzing for Linux-based IoT programs, but there are still some techniques that require further improvement in the future work. We discuss them from two aspects: binary static analysis and IoT program fuzzing.

1) Binary static analysis: We define the taint sink as the string or memory comparison library function in `uClibc` [12]. However, these comparison operation could be implemented in the binary itself without calling into library function. Meanwhile, some vendor could implement these operation in custom libraries with other names. For now, we do not take these situation into consideration which will miss some keywords for efficient fuzzing. In the future work, we need improve sinks identification mechanism to recall more taint sinks.

2) IoT firmware emulation: Since we utilize the emulation techniques of Firmadyne [8], the CPU architectures and IoT types we could perform greybox fuzzing are limited by Firmadyne. As a result, our system supports only ARM, MIPSEL, MIPSEB three CPU architectures and most of IoT types are cameras and routers. In the future, we will improve IoT firmware emulation techniques to support more diverse IoT types and CPU architectures.

V. Related work

1) Greybox fuzzing: For greybox fuzzing techniques. there are several ways to implement code coverage collection. One is dynamic instrumentation into binaries directly. WinAFL [17] is one of the representatives. The other is to utilize virtualization techniques. For example, AFL [4] instruments QEMU emulator code translation process to get code coverage. The third is to use hardware instrumentation techniques such as honggfuzz [18] which uses Intel Processor Trace techniques. Since Linux-based IoT programs require specific runtime system environment, all these tools could not execute IoT programs properly.

2) IoT fuzzing: In the aspect of IoT fuzzing, IoTFuzzer [6] performs blackbox fuzzing directly on the real device . It utilize communication protocol information from companion app to generate better test cases for fuzzing process that are more likely to trigger a bug. Unfortunately, most IoT devices do not have mobile apps. Muench et al. [19] implements the IoT device fuzzing by integrating a blackbox fuzzer Boofuzz [20] with different emulation configurations. Their emulation capability is provided by PANDA [21]. It mainly focuses on accurate detection of memory corruption vulnerabilities. TriforceAFL [9] follows AFL mechanism and integrate it with system mode QEMU to support the full system emulation and greybox fuzzing. It gets code coverage information by instrumenting QEMU system mode. Our work is built on top of TriforceAFL.

3) IoT Vulnerability discovery: Besides fuzzing, There are other static [22] [23] [24] [25] [26] and dynamic techniques [7] [8] [27] [28] that could help find vulnerabilities in IoT devices. Static techniques utilize program analysis techniques such as data flow analysis to find specific vulnerabilities or use code similarity comparison techniques to find homologous vulnerabilities. Dynamic works focus on the emulation of IoT firmwares or programs and using some dynamic analysis techniques such as taint analysis to find vulnerabilities.

VI. Conclusion

Recent start-of-the-art research works have solved IoT firmware fuzzing problems by achieving accurate IoT system emulation and integrating it with modern fuzzing tools. However, it is still not applicable and efficient for IoT programs fuzzing. Therefore, in our paper, we first perform static binary analysis to filter out `fuzz keywords` that could participate in input generation to improve fuzzing efficiency. We then reinforce the full emulation greybox fuzzer to support IoT program greybox fuzzing by achieving real-time program monitors which support accurate code coverage collection and new fork mode.

We evaluate the effectiveness of our system by discovering vulnerabilities in real world IoT programs. We also show efficiency of our work by comparing the discovered crashes and paths with random input generation fuzzing.

Acknowledgement

This work is partly supported by Guangdong Province Key Area R&D Program of China (Grant No.2019B010137004), National Natural Science Foundation of China (Grant No.U1636120), National Natural Science Foundation of China (Grant No.61702504), and Key Program of National Natural Science Foundation of China (Grant No.U1766215).

References

[1] "Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016," https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016, 2017.

[2] "Fuzzing," https://en.wikipedia.org/wiki/Fuzz_testing.

[3] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 1032–1043.

[4] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/.

[5] Y. Zheng, K. Cheng, Z. Li, S. Pan, H. Zhu, and L. Sun, "A lightweight method for accelerating discovery of taint-style vulnerabilities in embedded systems," in *International Conference on Information and Communications Security (ICICS)*, 2016, pp. 27–36.

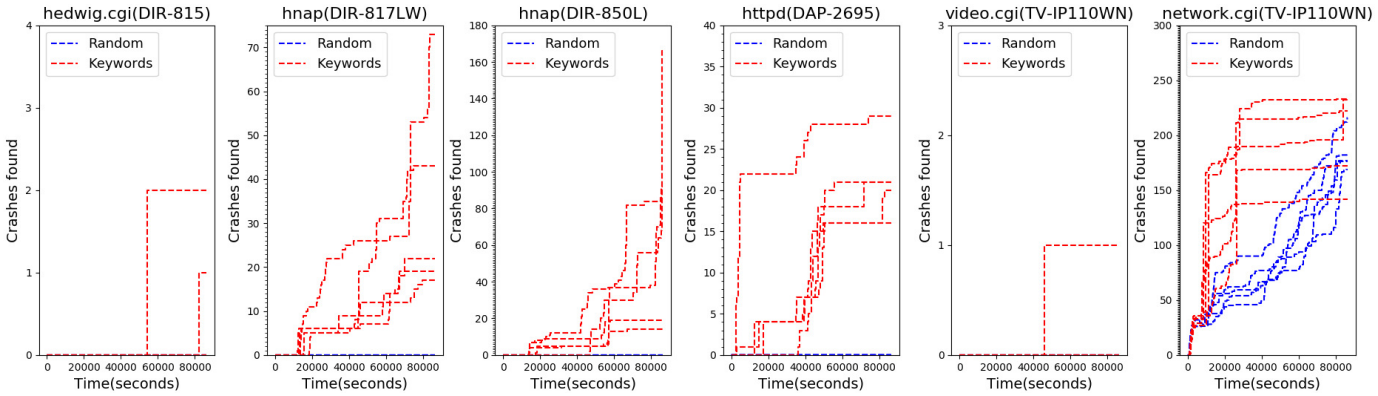

Fig. 3. Crashes found over time

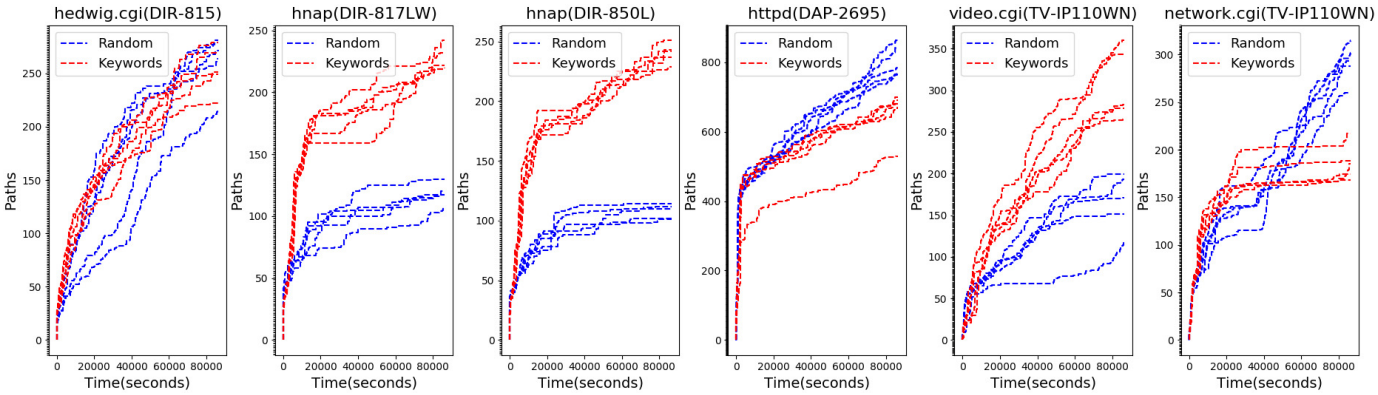

Fig. 4. Paths found over time

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

[7] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[8] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware," in *23rd Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[9] "TriforceAFL," https://github.com/nccgroup/TriforceAFL, 2017.

[10] "IDA: About - Hex-Ray," https://www.hex-rays.com/products/ida/index.shtml.

[11] "Firmware analysis tool," https://github.com/ReFirmLabs/binwalk.

[12] "uClibc - official site," https://uclibc.org/.

[13] "uClibc mirror," https://github.com/kraj/uClibc/tree/master/libc.

[14] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005, pp. 1032–1043.

[15] "IDAPython project for Hex-Ray's IDA Pro," https://github.com/idapython.

[16] "Firmadyne datasheet," https://cmu.app.boxcn.net/s/hnpvf1n72uccnhyfe307rc2nb9rfxmjp/folder/6601681737.

[17] I. Fratric, "A fork of afl for fuzzing windows binaries," https://github.com/googleprojectzero/winafl.

[18] "honggfuzz. a security oriented, feedback-driven, evolutionary, easy-to-use fuzzer with interesting analysis options," http://honggfuzz.com/.

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

[20] "Boofuzz," https://github.com/jtpereyda/boofuzz, 2016.

[21] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with panda," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, 2015.

[22] L. Cojocar, J. Zaddach, R. Verdult, H. Bos, A. Francillon, and D. Balzarotti, "PIE: Parser identification in embedded systems," in *Annual Computer Security Applications Conference (ACSAC)*, 2015, pp. 251–260.

[23] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

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

[25] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 480–491.

[26] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2017, pp. 363–376.

[27] M. Muench, A. Francillon, and D. Balzarotti, "Avatar^2: A multi-target orchestration platform," in *Workshop on Binary Analysis Research (BAR)*, 2018.

[28] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: A case study on embedded web interfaces," in *11th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016, pp. 437–448.