EM-Fuzz: Augmented Firmware Fuzzing via Memory Checking

Jian Gao, Yiwen Xu, Yu Jiang, Zhe Liu, Wanli Chang, Xun Jiao, and Jiaguang Sun

***Abstract*—Embedded systems are increasingly interconnected in the emerging application scenarios. Many of these applications are safety-critical, making it a high priority to ensure that the systems are free from malicious attacks. This work aims to detect vulnerabilities, that could be exploited by adversaries to compromise functional correctness, in the embedded firmware, which is challenging especially due to the absence of source code.**

In particular, we propose *EM-Fuzz*, a firmware vulnerability detection technique that tightly integrates fuzzing with real-time memory checking. Based on the memory instrumentation, the firmware fuzzing can not only be guided by the traditional branch coverage to generate high-quality seeds to explore hard-to-reach regions, but also by the recorded memory sensitive operations to continuously exercise sensitive regions which are prone to being attacked. More importantly, the instrumentation integrates real-time memory checkers to expose memory vulnerabilities, which is not well-supported by existing fuzzers without source code. The experiments on several real-world embedded firmware such as OpenSSL demonstrate that *EM-Fuzz* significantly improves the performance of state-of-the-art fuzzing tools such as AFL and AFLFast, with the coveragre improvements of 93.98% and 46.89% respectively. Furthermore, *EM-Fuzz* exposes a total of 23 vulnerabilities, with an average of about 7 hours per vulnerability. AFL and AFLFast together find 10 vulnerabilities, costing about 13 hours and 10 hours per vulnerability on average, respectively. Out of these 23 vulnerabilities, 16 are previously unknown and have been reported to the upstream product vendors, 7 of which have been assigned with unique CVE identifiers in the U.S. National Vulnerability Database.

***Index Terms*—Embedded firmware, Guided fuzzing, Memory checking, Vulnerability.**

I. Introduction

With the development of 5G technologies and beyond, the connectivity of embedded systems will get ever stronger in the emerging application scenarios, such as highly automated vehicles, domestic robots, and Industry 4.0. These applications are often safety-critical, strictly requiring functions to be correct. Therefore, being free of malicious attacks, e.g., malicious code injection and denial-of-service (DoS), needs to be ensured. The reality is however cruel. Cui *et al.* [1] scanned about 4 million embedded devices on the network and found 13.81% of the vulnerability rate. This work targets the embedded firmware, vulnerability detection of which is challenging, especially due to the absence of source code.

To enhance the ability of vulnerability exposure, fuzzing has become a research hotspot of firmware testing in recent years [2], [3], [4]. Peach [2] is a semi-automated cross-platform fuzzing framework which generates test inputs based on manually constructed input specifications. SRFuzzer [3] is a fully-automated fuzzing framework for fuzzing web servers of physical SOHO (small office/home office) routers. IoT-Fuzzer [4] uses the program logics of mobile app's to produce meaningful test cases for probing the target firmware. Although these approaches ensure the authenticity of the reported vulnerabilities, they belong to black-box fuzzing techniques that do not take into account execution feedback. Therefore the efficiency of vulnerability discovery is unsatisfactory.

With the greybox fuzzing technique showing efficient vulnerability discovery capabilities on programs with source code, the first thought is to use QEMU process emulation to perform similar greybox fuzzing on embedded firmware. In the absence of embedded devices, QEMU emulator indeed becomes the first choice for fuzzing firmware due to its relatively complete cross-architecture emulation capability and support for instrumentation to feed back branch coverage information. The binary versions of tools, such as AFL [5] and AFLFast [6], are representatives of such implementation, using branch coverage to guide the generation of test inputs continuously. However, we find in practice that their ability to deal with embedded firmware is greatly compromised. Two main reasons hinder the use of these tools in embedded scenarios.

One reason is that compared to fuzzing targets with source code, the time cost of using QEMU emulator to fuzz firmware is higher, and the number of test cases (also called throughput) that can be executed in a unit of time is 3–5× lower. Though existing fuzzers [5], [6] can be applied to embedded firmware, their ability to face low throughput is limited by inappropriate mutation frequency. For example, AFL's blind mutation strategy has difficulty mutating the meaningful seeds to efficiently reach hard-to-reach regions, and further resulting in the crash of the target under test.

Another reason is that when the firmware source code is not available, the existing fuzzers cannot implement source code instrumentation that can be combined with various sanitizers

Manuscript received April 17, 2020; revised June 17, 2020; accepted July 6, 2020. This article was presented in the International Conference on Embedded Software 2020 and appears as part of the ESWEEK-TCAD special issue. This work was sponsored in part by the NSFC Program under Grant 61527812, and in part by the National Science and Technology Major Project of China under Grant 2016ZX01038101.

J. Gao, Y. Xu, Y. Jiang and J. Sun are with the School of Software, Tsinghua University, Beijing National Research Center for Information Science and Technology, and Key Laboratory for Information System Security, Ministry of Education, Beijing 100084, China (e-mail: {gaojian094, xuyiwen14}@gmail.com, jiangyu198964@126.com, sunjg@tsinghua.edu.cn).

Z. Liu is with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China (e-mail: zhe.liu@nuaa.edu.cn.).

W. Chang is with the Department of Computer Science, University of York, UK (e-mail: wanli.chang@york.ac.uk).

X. Jiao is with the Department of Electrical and Computer Engineering, Villanova University, Villanova, PA 19085 USA (e-mail: xujiao@eng.ucsd.edu).

(e.g., AddressSanitizer, ThreadSanitizer) to check for memory corruption. According to statistics from NVD in the past three years, at least 9.82% of the included vulnerabilities were related to memory vulnerabilities, such as *use-after-free* and *buffer overflow*. This makes hidden memory vulnerabilities in the firmware less likely to be detected by an unassisted fuzzer. Although mature memory checkers, such as MemCheck [7] and Dr. Memory [8], have dominated the field of binary memory vulnerability checking through binary instrumentation, they have greater limitations on firmware. They are workable on the premise that they can be installed in the same running environment (e.g., same CPU architecture) as the binary under test. It would be impractical to install these memory checkers in an already-shipped firmware, so they basically do not work with the firmware. Furthermore, even if they are included in the firmware, executing arbitrary test cases obtained without guided fuzzing may be time-consuming and will not report vulnerabilities at all.

In the embedded scenario, although existing greybox fuzzers and memory checkers can exert a certain function, they are isolated. Combining fuzzing and memory checking to give full play to their strengths, we face the following challenges:

1) **Effective memory instrumentation.** The augmented fuzzer requires the necessary dynamic instrumentation of firmware, including integrating common types of memory detectors and recording memory sensitive operations in a unified way.
2) **Multi-architecture support.** Firmware CPU architectures present diversity due to different application scenarios of embedded devices. Using a unified running environment to handle multiple architectures of firmware fuzzing is eagerly welcomed.
3) **Coverage-guided and memory-guided collaboration.** Traditional branch guidance information is not enough to give full play to the capabilities of the combination. Discovering deeper and wider program paths and triggering as many memory vulnerability sensitive operations as possible is our desired goal.

To address the above challenges, we propose *EM-Fuzz* — an augmented firmware vulnerability detection technique via tightly integrating fuzzing with real-time memory checking. The memory checking module completes the firmware instrumentation task via memory hooking and library wrapping, which solves the challenge 1. In addition to integrating memory detectors to thoroughly identify memory vulnerability, it also monitors and records the detailed memory sensitive operations. Both the fuzzing and the memory checking in *EM-Fuzz* are implemented based on the QEMU emulator. Therefore, it is reasonable to use this unified running environment to address the multi-architecture support in challenge 2 and provide the capability for memory checking instrumentation in challenge 1. When emulating the firmware, the augmented QEMU emulator collects branch coverage information and memory sensitive operations, which together guide the mutation direction of the fuzzing and solve the challenge 3. The former guides the fuzzer to select seeds based on the traditional branch frequency, and mutate them with random and restricted mutation strategy, which ensures the depth and breadth of fuzzing. The latter guides the fuzzer to assign more mutation times to the seeds that have performed memory sensitive operations, which increases the chance that fragile regions can be covered more frequently.

For evaluation, we consider real-world embedded firmware programs, such as the IEC61850 protocol extracted from the substation automation system (SAS) firmware, and compare *EM-Fuzz* with two state-of-the-art fuzzers: AFL [5] and AFLFast [6]. The experimental results show that *EM-Fuzz* significantly improves performance. In particular, *EM-Fuzz* exposes a total of 23 vulnerabilities, while AFL and AFLFast expose 7 and 10, respectively. Combining the results from AFL and AFLFast, only 10 vulnerabilities are discovered. Out of the 23 vulnerabilities found by *EM-Fuzz*, 16 are previously unknown and have been reported to the upstream product vendors, 7 of which have been assigned unique CVE identifiers. As for the time cost, it takes *EM-Fuzz* 7 hours on average to discover one vulnerability, which is 54% and 70% of the time required by AFL and AFLFast, respectively.

In summary, this paper makes the following contributions:

- To our best knowledge, *EM-Fuzz* is the first tool that integrates the efficient greybox fuzzing and the memory checking to augment the capability of vulnerability discovery on the embedded firmware. While fuzzing has shown impressive performance on desktop applications, this work is a step for the embedded firmware without source code.
- We integrate 10 common types of memory detectors into *EM-Fuzz* through dynamic running instrumentation, which is able to enhance fuzzing to discover more previously unknown vulnerabilities on real-world firmware.
- We propose an optimized fuzzing strategy for embedded firmware. The traditional branch and the additional memory sensitive operation information guide fuzzing to explore hard-to-reach paths, and give fragile code areas more opportunities for exhausted memory checking.
- We apply *EM-Fuzz* on real-world embedded firmware, and discover many previously unknown vulnerabilities.

Paper Organization: Section II presents a motivating example. Section III details the design of *EM-Fuzz*. Section IV implements *EM-Fuzz* and reports the experimental results. Section V indicates future research efforts. Section VI is about related work, and Section VII concludes the paper.

II. Motivating Example

To show how *EM-Fuzz* can improve fuzzing performance and expose more vulnerabilities via memory checking instrumentation, we walk through the trimmed code snippet originated from the embedded firmware of IEC61850 shown in Listing 1. There are two vulnerabilities hidden in block D.

A. Necessity for Memory Instrumentation

Assuming the traditional fuzzer has a chance to explore the hard-to-reach region D, it still may not expose any memory vulnerability. In embedded scenarios without any assistance from memory checkers, fuzzers will consider that block D

is functioning properly since the overflow and the memory leak are not serious enough to crash the target firmware. As mentioned earlier, existing memory checkers (e.g., MemCheck [7], Dr. Memory [8]) require to be installed in the same environment as the firmware under test and run existing test inputs to detect memory vulnerabilities. And existing sanitizers (e.g., AddressSanitizer, ThreadSanitizer) require the original source code to accomplish the instrumentation, which is impractical in the testing of black-box embedded firmware.

```
1  #define FIXED_LENGTH 8
2  void parse_packet(char *buf){
3  ...
4  char* magic_values = getMagicValues(buf,offset=4,len=2);
5  char* func_codes = getFunctionCodes(buf,offset=8,len=2);
6  char* data, *other;
7  //BLOCK A
8  if (magic_values[0] == 0x88 && magic_values[1] == 0xB8){
9     //BLOCK B
10    if (func_codes[0] == 0x40 && func_codes[1] == 0x00){
11       //BLOCK C, TODO:
12    }else if(func_codes[0] == 0x7F && func_codes[1] == 0x02){
13       //BLOCK D
14       other = (char*)malloc(FIXED_LENGTH);
15       data = (char*)malloc(FIXED_LENGTH);
16       //assume data is "1234",len(data)=4
17       strcpy(data,getStatus(buf));
18       //assume other is "abcdefghijklmnopqr", len(other)=18
19       strcpy(other,getInfo(buf));
20       //BUG: buffer overflow, data="qr" now
21       storeToFile(file,data);
22       LOG(data,other);
23       //BUG: forget to free other and data
24    }else
25      //BLOCK E, TODO:
26 }else{
27    //BLOCK F
28    ERROR("Invalid_data_package");
29 }
30 ...
31 }
```

Listing 1. Motivating example that illustrates how *EM-Fuzz* addresses the problems of existing fuzzers.

In detail, Lines 14 and 15 of Listing 1 request 8 bytes of heap memory for the *other* and *data* variables, respectively. In the 32-bit OS environment, each variable is actually allocated 16 bytes of memory[1], and the memory space of the *data* is closely followed by that of the *other*. When the *data* memory is first filled with strings of less than 8 bytes in Line 17, and then the *other* memory is filled with strings of more than 16 bytes in Line 19, the contents of the *data* memory are contaminated by the *other* memory because the *data* memory is behind the *other* memory. Obviously, this buffer overflow issue that causes data integrity should be reported. In addition, Line 23 forgets to free the memory allocated in Line 14 and 15. If the function `parse_packet` is constantly called, the memory leak issue will make the embedded device run out of resources.

During fuzzing cross-architecture firmware, performing real-time memory checking of memory vulnerabilities (e.g., *use-after-free*, *buffer overflow*, *memory leak*) in a unified test environment would significantly improve the vulnerability detection performance. Without firmware source code, this goal is not well-supported by existing fuzzing techniques.

[1]It depends on the *malloc_chunk* data structure defined in glibc [9]. In the 32-bit Linux OS, the minimal size of the allocated memory chunk is 16 bytes.

B. Necessity For Efficient Fuzzing

After supporting memory checking instrumentation, detecting the buffer overflow vulnerability shown in Line 19 requires the test input *buf* to meet two conditions: 1) trigger the block D; 2) contain appropriate values for variables $data$ and $other$. Considering the low test input throughput of firmware fuzzing, the ideal goal should make block D explored more thoroughly.

The code snippet first obtains magic values and function codes from the data packet according to the fixed offsets 4–5 and 8–9 in Lines 4–5. Line 8 determines whether the magic value of 2 bytes is equal to 0xB888, and if not equal, it means that the data packet is invalid and gets filtered out in Line 26. The vast majority of test cases generated by representative fuzzers, such as AFL [5], AFLFast [6] and FairFuzz [10], try to maximize the traditional branch coverage. For example, AFL chooses a seed as the preference to perform mutations if it is the fastest and smallest input for any of the observed branches. FairFuzz performs the restricted mutation strategy on the selected seed to increase the probability of reaching the same hard-to-reach branch. However, its mutation strategy prefers to explore deeper branches rather than the wider ones, which leads to local convergence problem. Just considering the branch occurrence and ignoring the weight of the recorded memory sensitive operations may cause two problems: 1) branch BD would be reached too late; 2) the number of times that block C and block D are executed may be seriously unbalanced, where D is less frequently triggered.

EM-Fuzz aims that branch BD is reached as early as possible and block D is executed more times than block C, via the collaboration of the recorded memory sensitive operation guidance. The former can be achieved by optimizing the seed selection strategy and the mutation strategy of existing fuzzers. Each seed in the seed queue can be first selected with inverse probabilities to the occurrence frequency of the branch. This optimization increases the chance that the seed hitting the branch BE will be selected and mutated instead of being discarded early, which helps cover new rare branch BD early. In the mutation phase, if the selected seed hits rare branches and branches with many memory sensitive operations, *EM-Fuzz* applies the restricted mutation strategy to the seed to maintain the depth of path exploration. The latter can be implemented by adaptively updating seed mutation energy during the mutation stage. If the seed hits a code block that contains memory sensitive operations (e.g., `malloc`, `free`), it gives the seed more mutation energy to increase the number of times that such code block is covered with more test cases.

III. *EM-Fuzz* Design

As presented in Fig. 1, *EM-Fuzz* contains two major modules: memory checking instrumentation and guided fuzzing. By tightly integrating efficient fuzzing and real-time memory checking, analysts are able to locate serious security threats in the firmware quickly. The memory checking instrumentation module completes the firmware instrumentation task via memory hooking and library wrapping, which gains the ability to record memory sensitive operations to guide fuzzing and integrate common memory detectors to expose

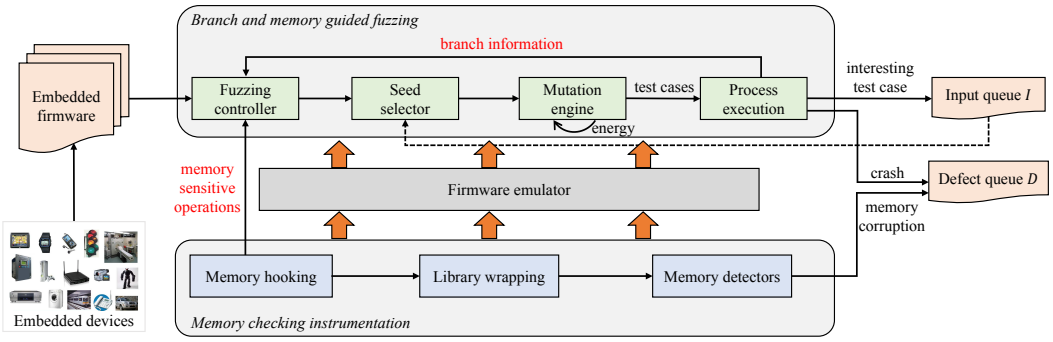


Fig. 1. The memory checking instrumentation module completes indispensable instrumentation, including recording memory sensitive operations to guide fuzzing and inserting 10 common types of memory detectors to identify memory vulnerability. The fuzzing module is guided by branch information to generate high-quality seeds to explore hard-to-reach regions, also by memory sensitive operations to continuously exercise regions that are prone to being attacked.

memory corruption vulnerabilities. Based on the firmware emulator with instrumentation capability, the fuzzing module seeks to generate high-quality test cases to explore hard-to-reach regions with branch coverage information guidance and trigger fragile regions that are prone to being attacked with memory sensitive operations guidance. As a result, the augmented firmware fuzzing not only ensures the depth and breadth of path exploration but also improves the ability of vulnerability exposure. During the entire fuzzing process, the two components share two test case queues, including an input queue I and a defect queue D.

A. Memory Checking Instrumentation

Memory checking instrumentation is responsible not only for feeding back memory sensitive operations to guide fuzzing but also for integrating memory detectors to check for memory vulnerabilities thoroughly. It contains memory hooking, library wrapping, and memory detector.

1) Memory Hooking: There are two main types of memory hooking in *EM-Fuzz*: the instruction hooking and the function hooking. The instruction hooking intercepts the *memory read* and *memory write* instructions to heap addresses. This is used to determine whether there are memory vulnerabilities. The function hooking intercepts memory-related library functions (e.g., `malloc`, `free`). Its purposes are to record memory sensitive operations in the execution trace, and to add prologue and epilogue actions.

For the instruction hooking, *EM-Fuzz* only hooks instructions that involve the heap memory access. Its purpose is to obtain the marked states of heap memory addresses so that memory detectors work. Therefore, we have to accurately distinguish the heap memory space from the entire virtual memory space. When *EM-Fuzz* loads the firmware into the emulator, it initializes the heap bottom address as the brk variable, denoted as $heapBottom = brk$. Whenever new heap memory space is allocated, the value of the brk variable grows to become the new heap top address. *EM-Fuzz* only hooks the instruction whose memory address access is in the range $heapBottom$ and brk.

For the function hooking, *EM-Fuzz* directly calculates the absolute virtual memory address of memory-operation related library functions. When the emulator executes to the `main` function, the library function `__libc_start_main` has been run and its start address is stored in the *.got.plt* section of the process. We can get the offset value of the `__libc_start_main` function from the *libc.so* library file. Indirectly, the start addresses of the memory-related library functions can be calculated according to the start address and the offset value of the known library function. The reason for not getting the start address of these functions directly from the *.got.plt* table is that other library functions (e.g., `printf`) may also call these functions through an offset in the same library file, at which point memory-related functions may have not been loaded in the *.got.plt* table due to the lazy binding mechanism.

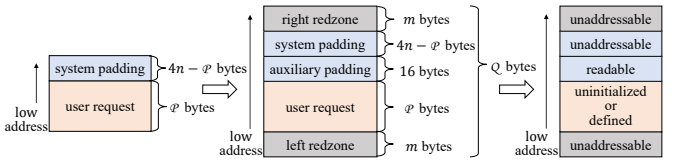
2) Library Wrapping: One of the main capabilities of *EM-Fuzz* is to support heap memory vulnerability detection for the firmware. With modifying the original memory-related library functions, we can implement memory detectors that require monitoring the entire heap memory state. *EM-Fuzz* achieves this goal through the library wrapping technique [8], which preserves the original heap layout.

We implement the library wrapping to add the prologue and the epilogue to each memory-related library function. The inserted prologues take precedence over the executions of memory-related library functions, which modifies the function parameters and assigns appropriate memory shadowing states (detailed in Section III-A3) to different memory regions. The epilogues are executed before the library functions return, which modifies corresponding return values. For heap memory of size $\mathcal{P}$, as requested by the user code in Fig. 2(a), the prologue re-layouts the virtual memory with the new memory size $\mathcal{Q}$ shown in Fig. 2(b). The memory size $\mathcal{P}$ requested by the user code (e.g., `malloc`) and the actual allocated memory size $\mathcal{Q}$ satisfy the following equation:

$$\begin{aligned} min \quad & \mathcal{Q} = m + \mathcal{P} + 16 + 4n - \mathcal{P} + m \\ s.t. \quad & (4n - \mathcal{P}) \geqslant 0 \\ & m = 8s, \quad s, n \in \mathcal{N}^* \end{aligned} \tag{1}$$

To give an example, if the user requests 5 bytes of heap memory using `malloc(5)`, the existence of the prologue causes the actual allocation of 40 bytes, of which 16 bytes are two redzones ($m = 8$ by default), 16 bytes are auxiliary padding, 3 bytes are system padding, and 5 bytes are available to the user. Assuming that the start address of the allocated 40 bytes is

0x804b008, the epilogue sets the return address to 0x804b010 before the `malloc` returns.



(a) Memory layout of the original allocation-related library functions (b) Memory layout of the new allocation-related library functions with epilogues (c) Illustration of marking memory areas with different states

Fig. 2. Memory layout and corresponding accessible states of the allocation-related standard library functions.

3) Memory Detectors: *Memory detectors* must track the memory usage states to detect memory vulnerabilities. *EM-Fuzz* uses memory shadowing to mark the state of each heap memory byte. The memory shadowing of each byte can only be one of the following four items at a certain time:

- *unaddressable*: illegal heap memory that is not allowed to be accessed by programs within the firmware.
- *uninitialized*: addressable heap memory that has been allocated but has not been written.
- *defined*: addressable heap memory that has been allocated and has been written.
- *readable*: addressable heap memory that is only readable in the string processing library functions.

Fig. 2(b) and Fig. 2(c) illustrate the correspondence between different heap memory regions and memory shadowing. Unaddressable memory region includes not only the redzone and the system padding, but also other memory addresses that are not allocated. Memory region that satisfies the size of the user code request can be either uninitialized or defined, depending on whether the user code assigns values to them. The auxiliary padding region is marked as readable, which is only usable in string processing library functions.

TABLE I
HEAP MEMORY VULNERABILITY SUPPORTED BY *EM-Fuzz*

Memory detectors	Implementation principle
buffer overflow	write to the system padding, auxiliary padding and right redzone regions marked as unaddressable or readable state
buffer over-read	read from the system padding, auxiliary padding and right redzone regions marked as unaddressable state
buffer underflow	write to the left redzone marked as unaddressable state
buffer under-read	read from the left redzone marked as unaddressable state
double free	mark the freed memory address for the first time, then monitor the second `free` calling of the same address
use-after-free	mark the freed memory address for the first time, then access address marked as the uninitialized or defined state
wild free	judge whether the address to be freed is the start address of the heap memory
uninitialized access	access memory address with the uninitialized shadowing
read invalid memory	read from memory address marked as the unaddressable state
memory leak	hook the `exit` system call, reports the address and size of the memory that are not in the unaddressable state

Table I lists ten common types of heap memory vulnerabilities supported by *EM-Fuzz* for checking. Each type of vulnerability corresponds to one memory detector in *EM-Fuzz*, as shown in Column 1. Column 2 explains the simple implementation principle in *EM-Fuzz*. For example, *buffer overflow* means writing data to an adjacent memory location that overruns the buffer's boundary. *EM-Fuzz* detects this behavior by determining whether instructions write data to the system padding, auxiliary padding, and right redzone regions marked as the unaddressable or readable state. The memory leak detector is a bit special, it hooks the `exit` system call and counts memory bytes that are not in the unaddressable state.

If there is no readable state, reading data from heap memory in the string handling functions is often accompanied by the buffer over-read false positive. Taking the *strlen* function as an example, in order to improve efficiency, it may not read the character from heap memory byte by byte and determine if there is a terminator '\0'. Different implementation versions of the *strlen* function can read one word, double words, or even 16 bytes (using the SSE instruction set) at a time, find the terminator from the front to back and return the string length. To eliminate this type of false positives, we used auxiliary padding marked as "readable" to filter out buffer over-reads occurring in the string processing library functions.

B. Guided Fuzzing

The overall procedure of the additional memory sensitive operation guided fuzzing is presented in Algorithm 1. Lines 1–17 are the main pseudocode implementation that will continuously execute until the timeout is reached or the fuzzing is aborted purposely, where improvements to existing fuzzers are specifically marked by gray boxes. The optimization is designed to more easily explore hard-to-reach deep paths without losing the breadth of path exploration and give fragile code areas more opportunities for exhausted memory checking. The details of each step are described below.

Algorithm 1: Memory sensitive fuzzing algorithm

```
Input: S: initial seed set
Output: I: test input queue that participates in mutation
Output: D: test input queue that makes firmware crash
1  Function DualGuidedFuzzing():
2      I = S, D = ∅
3      while !isTimeout do
4          seed = ChooseSeed(I) //branch frequency based skip rule
5          recordedMemoryOps = getRecordedMemoryOps(seed)
6          energy = CalculateMutation(seed, recordedMemoryOps)
           //consider memory sensitive operations and selected frequency
7          rareBranchFlag = IsHitRareBranch(seed)
8          memSensitiveFlag = IsMemSensitiveBranch(seed)
9          for i from 1 to energy do
10             if rareBranchFlag or memSensitiveFlag then
11                 mutatedCase = RestrictedMutation(i, seed)
                   //keep fixed offsets of the selected seed immutable
12             else
13                 mutatedCase = RandomMutation(seed)
14             end
15             ProcessExecute(mutatedCase)
16         end
17     end
18 End Function
```

1) Fuzzing Controller: As the basis of the entire fuzzing process, *fuzzing controller* receives feedback after executing each test case derived from the mutation stage, including

branch coverage and memory sensitive operations. It keeps track of each branch that has been discovered and records which test cases have covered the branch. Meanwhile, it is responsible for updating the global threshold used to dynamically separate common and rare branches. It also records which memory sensitive operations (e.g., *malloc*, *free*) are included in the execution trace of each test case. These information provides sufficient support for subsequent seed selection, energy assignment, and seed mutation.

2) Seed Selector: To cover hard-to-reach branches faster and earlier, *EM-Fuzz* uses the seed skip principle based on branch frequency to select seeds for mutation, which effectively solves the problem that existing fuzzers [5], [6], [10] may leave some hard-to-cover branches late or not discovered.

We use the term *hit count* to indicate the number of times that a branch has been executed. *EM-Fuzz* initially performs one round of mutation on each seed of the input queue I to get the map $\mathcal{M}$ recording the hit count of each branch, and to acquire all observed branches set $\mathcal{B}$. Each time a newly generated seed is executed, the values of $\mathcal{M}$ and $\mathcal{B}$ are updated. Let n be the number of branches in seed $\mathcal{S} \in I$, b_j be the j^{th} branch, and mb represent the branch with the minimum hit count. We use formula (2) to represent the probability that seed $\mathcal{S}$ will be skipped based on the execution frequency of each branch, where γ is a balanced constant.

$$\mathcal{P}(\mathcal{S}) = \left(1 - \frac{\mathcal{M}^{-1}(mb)}{\sum_{j=1}^{n} \mathcal{M}^{-1}(b_j)}\right) \cdot \gamma \tag{2}$$

From the above formula, we know that seeds that execute rare branches have a lower probability of being skipped. It ensures that hard-to-reach regions that have been observed can still be more easily covered. However, it also gives the seeds that are not preferred in other fuzzers a certain probability of being selected, which increases the chance to cover other hard-to-reach regions that have not been seen before. In a similar way, we can increase the skipping probability of seeds that cover fewer memory sensitive operations.

3) Seed Mutator: This step consists of two phases: calculate mutation energy and apply mutation strategy. They directly determine when *EM-Fuzz* can achieve the maximum branch coverage and how many vulnerabilities can be detected.

Mutation energy. *EM-Fuzz* takes the branch hit count and the number of memory sensitive operations into account to give different seeds the appropriate mutation energy on the basis of AFLFast [6]. The firmware emulator in *EM-Fuzz* is implemented with the function of memory checking instrumentation. During processing each test case, it records several categories of memory sensitive operations, including allocation functions (e.g., `malloc`, `calloc`), movement functions (e.g., `memmove`, `strcpy`), comparison functions (e.g., `strcmp`, `memcmp`), release function (e.g., `free`), etc., which are library functions easy to cause vulnerabilities.

Let $c(\mathcal{S})$ denote the number of times that the seed $\mathcal{S}$ has been chosen from the input queue I, $f(\mathcal{S})$ represent the number of fragile library functions in the execution trace of the seed $\mathcal{S}$, mb be the branch with minimum hit count among the branches covered by seed S and $\mathcal{M}(mb)$ be its hit count. The new mutation energy $\mathcal{E}$ is computed as the formula (3):

$$\mathcal{E} = min\left(\frac{\mathcal{E}_o}{\alpha} \cdot \frac{2^{c(\mathcal{S})} \cdot max(1, f(\mathcal{S}))}{\mathcal{M}(mb)}, \mathcal{U}\right) \tag{3}$$

where $\alpha > 1$ is a constant that balances the relationship between original AFL's energy $\mathcal{E}_o$ and new *EM-Fuzz*'s energy $\mathcal{E}$, $\mathcal{U}$ is an upper bound on the number of mutations. It improves AFLFast's exponential energy allocation strategy by additionally considering memory sensitive operations, which increases the frequency of testing fragile code regions.

Mutation strategy. Drawing on the partial ideas of FairFuzz and in combination with our proposed seed selection strategy, *EM-Fuzz* applies different mutation strategies to different seeds, where the restricted mutation strategy and the random mutation strategy. Line 7 in Algorithm 1 first determines whether the selected seed $\mathcal{S}$ covers rare branches. If so, it performs the restricted mutation that continues to cover hard-to-reach or memory sensitive branches. Otherwise, it performs the random mutation that covers a wider range of new branches. Let min_hit denote the minimum hit count for all branches in $\mathcal{B}$. As new test cases are executed, the hit count of each branch also continues to increase, and only the branches $b \in \mathcal{B}$ with the hit count $\mathcal{M}(b)$ that satisfy formula (4) belong to rare branch rb.

$$\mathcal{M}(b) < rarity\_cutoff \tag{4}$$

where $rarity\_cutoff$ is the threshold separating common and rare branches, and its real-time value is $2^{\lceil log_2 min\_hint \rceil}$.

If the selected seed hits a rare branch or a memory sensitive branch, *EM-Fuzz* applies the restricted mutation strategy. It first determines which fixed offsets of the selected seeds should remain unchanged. Therefore, *EM-Fuzz* performs a round of mutations on each byte of the seed $\mathcal{S}$ by trying three types of operations: insertion, deletion, and replacement. If any mutation operation on the fixed offsets makes newly obtained test cases no longer hit the same rare branch as the original seed $\mathcal{S}$, then these fixed offsets belong to immutable key bytes, and vice versa. These immutable bytes will remain unchanged during the restricted mutation strategy, and *EM-Fuzz* performs random mutation strategy on other offsets of the seed based on the assigned mutation energy. For the selected seed that does not hit a rare branch or a memory sensitive branch, *EM-Fuzz* follows the original random mutation strategy of AFL to increase the probability of finding new branches.

4) Process Execution: When executing each test case, *EM-Fuzz* determines whether it causes the firmware to crash or corrupt. If so, *EM-Fuzz* adds such test case to the defect queue D, otherwise, it measures whether an interesting branch ib is hit. One of the following two conditions makes a seed interesting: 1) exercise a new branch that is not observed in the previous branch set $\mathcal{B}$, presented as $ib \notin B$. 2) the number of times that branch ib is executed by test case $\mathcal{T}$ is significantly different from the number of executions by previous any test case $\mathcal{T}'$ in input queue I, formulated as

$$\forall \mathcal{T}' \in I, \lfloor log_2(\mathcal{HS}(\mathcal{T}, ib)) \rfloor \neq \lfloor log_2(\mathcal{HS}(\mathcal{T}', ib)) \rfloor \tag{5}$$

where $\mathcal{HS}(\mathcal{T}, ib)$ represents the number of times that branch ib is exercised by the test input $\mathcal{T}$, $\lfloor log_2(\cdot) \rfloor$ is the floor function of the logarithm to the base of two.

IV. Implementation and Evaluation

EM-Fuzz's firmware emulator is augmented with memory checking instrumentation based on the QEMU emulation platform [11]. It mainly enhances the original QEMU in three parts to achieve dynamic runtime instrumentation. First, when QEMU starts to run a firmware binary, it loads shared memorys to record branch coverage information and the number of memory sensitive operations. Then, before each translation block is executed, it instruments a value to uniquely identify each translation block similar to AFL [5]. Therefore, when a test case is processed, the transition relationship between two contiguous translation blocks and the memory sensitive operation information can be updated to the corresponding shared memory. Finally, when a memory-related library function call is encountered during the execution of translation blocks, it wraps the original library function and sets the appropriate memory shadowing as described in Section III-A2. If memory write/read instructions to heap addresses are encountered, it calls back ten types of memory detectors embedded in the enhanced QEMU to detect heap memory defects.

The augmented QEMU emulator works together with the guided fuzzing module. Both of them are controlled by the *synchronous controller* that is built on top of AFL [5]. To reduce the impact of low throughput on firmware fuzzing, we depend on the vital feedback from memory checking instrumentation to get the memory-sensitive operations of the closed-source firmware, and strengthen three functions of the original AFL: allow each seed in the queue I to participate in mutation according to the seed skipping principle, assign mutation energy based on branch hit count and memory sensitive operations, and apply appropriate mutation strategies for different selected seeds.

We evaluate whether *EM-Fuzz* is able to address the bottlenecks in testing real-world embedded firmware and efficiently discover multiple types of vulnerabilities. We would like to answer the following three research questions:

- *RQ1: Efficiency in branch discovery.* Is *EM-Fuzz* efficient in improving branch coverage of firmware?
- *RQ2: Effectiveness in vulnerability exposure.* Is *EM-Fuzz* effective in exposing multiple types of vulnerabilities?
- *RQ3: Performance of memory detectors.* What performance can the detectors of *EM-Fuzz* achieve against cross-architectures firmware?

A. Experiments Setup

We evaluate the performance of *EM-Fuzz* on six embedded programs, including the ones that were also studied in other works [6], [12]. They were from two types of embedded device firmware as listed in Table II. The first type was protocol programs (ICCP [13], IEC104 [14], and IEC61850 [15]) extracted from substation automation devices responsible for time-critical information exchange between the control center and remote terminal units (RTUs) in the power industry. The second type contains three widely-used programs (security communication toolkit *OpenSSL*, data processing libraries *HTSlib* and *MXML*) extracted from the *OpenWRT* router firmware. We compare *EM-Fuzz* with two state-of-the-art fuzzers to answer RQ1 and RQ2, including AFL [5], and AFLFast [6]. RQ3 is responded by comparing with the widely-used binary memory detector Dr. Memory [8].

TABLE II
Embedded programs extracted from real-world firmware for experimental evaluation

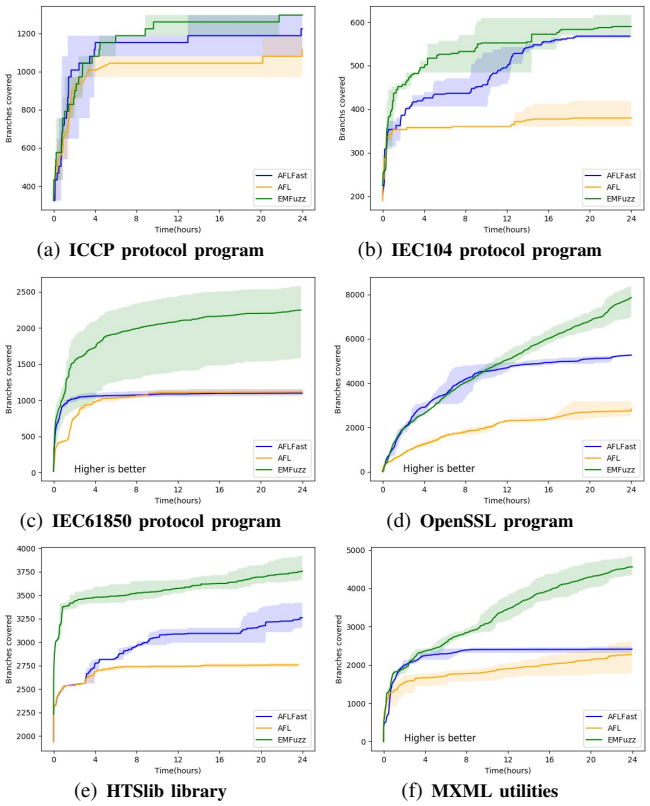
Program	Firmware Type	Version	Architecture
ICCP	Substation automation system (SAS)	V1.5	X86
IEC104		V1.0.0	X86
IEC61850		V1.3.1	ARM32
OpenSSL	Router	V1.0.1b	ARM32
HTSlib		V1.8	
MXML		V2.12	

We run the experiments on a 64-bit machine with 36 cores (Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz), 128GB of main memory, and Ubuntu 16.04.6 LTS as the host OS. For each tool, we run each firmware program ten times for 24 hours. We repeat each program ten times with the same seed file to reduce the randomness of fuzzing. The results reported in subsequent experiments are summarized after ten independent runs for each target tested.

B. Efficiency in Branch Discovery

Here we answer the RQ1 about what coverage *EM-Fuzz* can achieve within the limited time. We chose the number of branches covered by the 24-hour fuzzing as a guideline for evaluating the efficiency. Fig. 3 depicts the trend of branch discovery, indicating the total number of branch tuples found by each tool on average at each time point over ten 24-hour runs. On all fuzzed programs, *EM-Fuzz* eventually achieves a higher number of branches covered, with the *EM-Fuzz* curve above the other curves in Fig. 3. *EM-Fuzz* covers an average of 3,386 branches on all six firmware programs, which is 93.98% and 46.89% more than that of AFL and AFLFast, respectively.

For the specific OpenSSL program, *EM-Fuzz* covers as many as 7,862 branches, which is 1.78× more than AFL and 0.49× more than AFLFast. We can also see that the curves of AFL and AFLFast on four programs eventually become stable and no longer increase, which is referred to as "saturation". Excluding the OpenSSL and MXML programs, AFL saturates after approximately 5 hours of fuzzing on the other 4 programs. Excluding the IEC104, OpenSSL and HTSlib programs, AFLFast reaches saturation at approximately 16 hours. However, *EM-Fuzz* still presents a trend toward an increase in the number of branches covered on all programs in the 24-hour fuzzing. The reason is that the mutation strategy of *EM-Fuzz* moves toward the earlier coverage of rare branches, making it easier to explore deeper program paths in a short period of time. For the IEC61850 protocol program, its valid input may contain several specific fields, in which different fields match different application functions, and *EM-Fuzz* generates more interesting test inputs to cover a series of branches corresponding to these functions. The experiment results imply *EM-Fuzz*'s excellence not only in the number of branch discovery but also in the speed boost.



(a) **ICCP protocol program** (b) **IEC104 protocol program**

(c) **IEC61850 protocol program** (d) **OpenSSL program**

(e) **HTSlib library** (f) **MXML utilities**

Fig. 3. Number of branches covered by contrast tools averaged over 10 runs.

The ICCP and IEC104 protocal programs showed less improvement in branch coverage than that of the other four programs. Two reasons contributed to this situation. First, we conducted the reverse analysis of the binary of the six programs, and found that the size of these two programs was smaller, and the number of branches was significantly less than the other four programs. Therefore, even a small increase in branch coverage is not trivial. And the prior study [16] showed that even small increases in branch coverage can obtain more defects finding capability after a certain amount of coverage is achieved. Second, as time progresses, rare branches gradually became common branches and the number of other rare branches rediscovered was inadequate. Therefore, *EM-Fuzz*'s mutation strategy cannot give full play to its value.

C. Effectiveness in Vulnerability Exposure

Here we answer the RQ2 and focus on whether *EM-Fuzz* can expose multiple types of vulnerabilities in firmware effectively. We choose the number and time for illustration.

1) Total Number of Vulnerabilities: The ability to expose vulnerabilities is an important indicator of fuzzers. Therefore, we count the number of vulnerabilities that have been confirmed officially, including reproducing existing known vulnerabilities and discovering previously unknown ones. After running each program ten times with each tool, we fill the total number of officially confirmed vulnerabilities discovered by each tool into Table III. Column 1 refers to the firmware names. Column 2 represents the identifier information for each vulnerability, where 'CVE-XXX' means that the vulnerabilities have been officially confirmed and assigned with the unique CVE identifier in the U.S. national vulnerability database [17], and 'Bug-XXX' means that the vulnerabilities have been reported the corresponding product vendors and officially confirmed, the previously unknown vulnerabilities are marked with ⋆. Columns 3, 5, and 7 indicate whether the three tools in comparison expose each vulnerability, and if so, the corresponding cell is filled with the check mark (✓); otherwise, it is empty.

From Table III, we can see that *EM-Fuzz* discovers a total of 23 vulnerabilities, which are 16 and 13 more than AFL and AFLFast, respectively. The 16 of the 23 vulnerabilities found by *EM-Fuzz* are previously unknown. The two compared tools together found only 6 of the 16 previously unknown vulnerabilities. Among the four vulnerabilities in the ICCP protocol program, AFL and AFLFast only found one segmentation fault vulnerability (shortened to SIGSEGV). A similar situation also exists for the IEC104 protocol program. The improvement is mainly brought by the real-time memory checker and the memory sensitive operation information acquired by the extra memory instrumentation. It can not only expose those vulnerabilities that would not result in crashes, but also helps to learn the energy of test input and generate test inputs that explore deeper and fragile program regions more often. For example, neither AFL nor AFLFast exposes the SIGSEGV vulnerability identified by Bug-2019-0921 in the IEC104 protocol program. Likewise, they found no vulnerabilities in the HTSlib library that accesses high-throughput sequencing data in ten runs. These show that *EM-Fuzz* has a better ability to detect vulnerabilities in embedded firmware.

2) Types of Vulnerabilities Found: We analyze the ability of each tool to expose each type of vulnerability. Table IV lists the statistical results, where each vulnerability belongs to only one specific type and is not shared with other types. From Table IV, we can see that *EM-Fuzz* performs better in reporting the number of vulnerabilities in each category. The first six types of vulnerabilities discovered by *EM-Fuzz* accounts for 12 of a total of 23 vulnerabilities, with a ratio of 52.17%. Neither AFL nor AFLFast can report any vulnerabilities related to heap out-of-bounds access. After analyzing these 12 vulnerabilities, we find that seven of them only slightly cross the corresponding heap boundary. It is not enough to cause the firmware to crash, so AFL and AFLFast can not expose them. With the implemented heap memory detectors, *EM-Fuzz* can accurately report heap memory issues in firmware programs.

Aside from heap memory vulnerabilities, we focus on the last three types of vulnerabilities in Table IV. As they would easily result in crashes, each tool in the comparison performs well. AFL and AFLFast expose 7 and 10 vulnerabilities, reaching only 63.64% and 90.91% of these three types of vulnerabilities that are found by *EM-Fuzz*, respectively. The reason is similar to the previous illustration that *EM-Fuzz* applies the optimized fuzzing strategy for firmware. The traditional branch and the additional memory sensitive operation information guide the fuzzing to more easily explore hard-to-reach deep paths, and give fragile code areas more opportunities for exhaustive fuzzing. These statistics show that the traditional fuzzing approaches perform well in detecting

TABLE III
LIST OF VULNERABILITIES FOUND BY EACH TOOL WITHIN 24 HOURS

Program	Identifier	Type	AFL		AFLFast		EM-Fuzz	
			Found(?)	Time(h)	Found(?)	Time(h)	Found(?)	Time(h)
ICCP	Bug-2019-0923⋆	segmentation fault	✓	8	✓	7	✓	6
	Bug-2019-0925⋆	wild free					✓	5
	Bug-2019-0928⋆	buffer over-read					✓	2
	Bug-2019-1003⋆	buffer overflow					✓	12
IEC104	Bug-2019-0912⋆	buffer over-read					✓	5
	Bug-2019-0915⋆	stack overflow			✓	16	✓	10
	Bug-2019-0918⋆	segmentation fault	✓	4	✓	4	✓	5
	Bug-2019-0921⋆	segmentation fault			✓	8	✓	6
IEC61850	CVE-2018-18834	buffer overflow					✓	3
	CVE-2018-18937	NULL pointer dereference	✓	10	✓	9	✓	7
	CVE-2018-19093	segmentation fault	✓	15	✓	14	✓	11
	CVE-2018-19185	buffer overflow					✓	2
	CVE-2018-19121	segmentation fault	✓	22	✓	17	✓	8
	CVE-2018-19122	NULL pointer dereference			✓	21	✓	13
	CVE-2019-6136⋆	segmentation fault					✓	13
OpenSSL	CVE-2016-2108	buffer underflow					✓	13
	Bug-2019-0824⋆	stack overflow	✓	6	✓	5	✓	5
HTSlib	CVE-2018-13843⋆	memory leak					✓	7
	CVE-2018-13844⋆	memory leak					✓	8
	CVE-2018-13845⋆	buffer over-read					✓	5
MXML	CVE-2018-19764⋆	stack overflow	✓	19	✓	7	✓	3
	CVE-2018-20004⋆	use-after-free					✓	6
	CVE-2018-20005⋆	memory leak					✓	5
Total			7	-	10	-	23	-

* Previously unknown vulnerabilities are marked with ⋆. Vulnerabilities found by each tool over ten 24-hour runs are marked with ✓.

TABLE IV
THE NUMBER OF EACH TYPE OF VULNERABILITIES FOUND BY THE CONTRAST TOOLS ON THE SIX FIRMWARE PROGRAMS

Catagory	AFL	AFLFast	EM-Fuzz
buffer overflow	0	0	3
buffer underflow	0	0	1
buffer over-read	0	0	3
wild free	0	0	1
use after free	0	0	1
memory leak	0	0	3
NULL pointer dereference	1	2	2
stack overflow	2	3	3
segmentation fault	4	5	6
Total	**7**	**10**	**23**

vulnerabilities that cause the target to crash. In contrast, with real-time memory checking of *EM-Fuzz*, we can detect more vulnerabilities in embedded firmware.

3) Time of Vulnerability Discovery: Although *EM-Fuzz* can cover more branches faster and find more vulnerabilities, we want to know how long it takes to expose a vulnerability. Columns '*Time*' in Table III lists the minimum time at which the three tools in comparison expose each vulnerability in ten runs. The empty cell indicates that the corresponding tool does not find the vulnerability in ten 24-hour experiments. We can see that it took *EM-Fuzz* 13 hours to expose all 23 vulnerabilities, of which 17 of the 23 vulnerabilities could be exposed within 10 hours. On the whole, AFL and AFFLFast require 1.71× and 1.54× more time on average than *EM-Fuzz* to expose vulnerabilities, respectively. For example, AFL requires 22 hours to expose the CVE-2018-19121 vulnerability, and AFLFast requires 21 hours to expose the CVE-2018-19122 vulnerability. In contrast, the CVE-2018-19185 vulnerability in the IEC61850 protocol program can be reported by *EM-Fuzz* in 2 hours.

4) EM-Fuzz's Time Overhead: In the same running environment, we selected four open-source projects (Coreutils [18], libpng [19], zlib [20], yaml-cpp [21]) that were also studied in other works [6], [12] to evaluate the time overhead introduced by *EM-Fuzz*. Using the QEMU emulator without memory checking instrumentation to run these projects costs about 3–5× more than running them directly on the native processor. When the memory checking instrumentation is turned on, the time cost will increase by an average of 1.5× again to 4.5–6.5×.

D. Performance of Memory Detectors

We answer the RQ3 about whether *EM-Fuzz* can accurately detect heap memory vulnerabilities against diverse architecture firmware in a unified test environment. To illustrate the accurateness and versatility, we compare *EM-Fuzz*'s memory checker module with the state-of-the-art memory checker Dr. Memory [8], both of which run on the server with the X64 architecture as described in Section IV-A. Since Dr. memory does not have the ability to generate test cases automatically, we save all test cases derived from the fuzzing phase of *EM-Fuzz*. When using Dr. memory to test firmware, these test cases are input in turn. Because the main ability of Dr. Memory is to detect heap memory vulnerabilities, therefore, for a better and fairer comparison with Dr. Memory, only heap memory vulnerabilities are counted here. All detected heap memory vulnerabilities are summarized and deduplicated, and the results are listed in Table V.

EM-Fuzz can detect 12 heap memory vulnerabilities on all six firmware programs, while Dr. Memory can only detect 4 heap memory vulnerabilities on X86-based programs. *EM-Fuzz*, like the well-known memory checker Dr. Memory, accurately detects 3 and 1 heap memory vulnerabilities in ICCP and IEC104 programs, respectively. This implies that

TABLE V
NUMBER OF HEAP MEMORY VULNERABILITIES DETECTED

Program	Dr. Memory$^{EM-Fuzz}$	EM-Fuzz
ICCP	3	3
IEC104	1	1
IEC61850	-	2
OpenSSL	-	1
HTSlib	-	3
MXML	-	2
Total	**4**	**12**

EM-Fuzz can accurately identify heap memory vulnerabilities in firmware. The reason Dr. Memory cannot work on the other four programs is that DynamoRIO [22], its instrumentation framework, also requires running on the ARM-based machine to detect ARM architecture firmware. Different from it, *EM-Fuzz* can use a unified test environment, such as on the X86 machine, to detect the memory vulnerability of multi-architecture firmware by using the emulator implemented with memory checking instrumentation. This feature makes *EM-Fuzz* more efficient and practical, which can use resource-rich servers to fuzz cross-architecture firmware and additionally expose memory vulnerabilities hidden in them.

E. Real Vulnerability Case Study

During the experiment, *EM-Fuzz* exposes two serious vulnerabilities that can easily cause denial-of-service (DoS), CVE-2016-2108 on OpenSSL and Bug-2019-0921 on IEC104. Taking the CVE-2016-2108 vulnerability as an example, its CVSS (Common Vulnerability Scoring System) score [23] is the highest of 10.0, meaning it is a vulnerability prone to catastrophic consequences. Consider the test input shown in Fig. 4, which is the data object description structure that follows the Abstract Syntax Notation One standard (ASN.1). In order to implement network communication, the ASN.1 parser *asn1parse* encodes the left data structure of Fig. 4 into the serialized binary stream based on specific encoding rules, such as DER, PEM. As shown in the code snippet of Fig. 4, existing fuzzers have difficulty generating test inputs to trigger the branch condition of Line 422 that contains the vulnerability implementation. When the parser deserializes the binary stream, if the value of the INTEGER type is '0x-0', the parser produces a buffer underflow with an out-of-bounds write in the `i2c_ASN1_INTEGER` function. The immediate consequence of buffer underflow is that the chunk field of the allocated heap is inadvertently modified, which causes the parser to crash when releasing the corrupted heap memory. The deserialization and serialization process allows remote attackers to corrupt memory, indirectly leading to a denial-of-service vulnerability.

Only *EM-Fuzz* has ever reproduced the high-risk vulnerability during the experiment. With *EM-Fuzz*'s fuzzing strategy, it is easier to generate well-formed test inputs and assign such test inputs more mutation energy, which increases the probability that other contents except '0x-0' are correct. It will keep "sequence" and "INTEGER" immutable; otherwise, the parsing logic will enter the error handling branch.

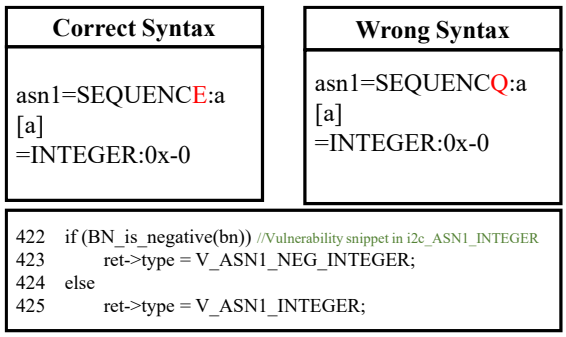


Fig. 4. Examples of data object description structures to trigger the crash and the corresponding vulnerable code snippet in OpenSSL.

V. DISCUSSION

In this section, we discuss the limitations of *EM-Fuzz* and indicate future research efforts. The first possible threat is the type of vulnerability *EM-Fuzz* can discover. Although *EM-Fuzz* has integrated the efficient greybox fuzzing and the memory checking to augment the capability of vulnerability discovery on the embedded firmware, it mainly focuses on finding the following categories of vulnerabilities in firmware, including program crashes, execution timeouts, silent memory corruptions, and some logical bugs (e.g., an infinite loop). However, *EM-Fuzz* does not show sufficient superiority to web security-related vulnerabilities in firmware, such as XSS and command injection. The main reason is that the current implementation does not include suitable monitors to identify such vulnerabilities that usually do not cause crashes. A feasible solution is to set up a proxy-based server in the local network to determine whether the monitor server is accessed by the firmware, and thus identifying XSS and command injection vulnerabilities [3].

Another threat is that fuzzinging may fall into a state where branch coverage is saturated, even though we have proposed several optimized fuzzing strategies for embedded firmware to better balance the depth and breadth of branch discovery. However, as the fuzzing time progresses, there may be some special filtering mechanisms in the firmware code (e.g., CRC checksum and hash mapping) causing the fuzzing stuck. That means it needs to take lots of efforts to discover new branches. When no new branch is found after the threshold is exceeded, a feasible solution is to use symbolic execution to generate test cases to bypass certain branches that are difficult to cover [24].

VI. RELATED WORK

A. Binary Fuzzing Techniques

Black-box [2], [25], [26] and greybox [5], [6], [10], [12], [27], [28], [29] techniques play a dominant role in the field of binary fuzzing when the source code is not available. Mutation-based black-box fuzzers [25], [26] mutate the corresponding proportion of bits within the test inputs according to the specified fuzzer ratio. SymFuzz [26] adaptively sets the fuzzer ratio by determining dependencies between the execution trace and the bit positions of a test input. Generation-based black-box fuzzers [2] generate inputs from scratch based

on specifications. Peach [2] relies on the predefined data model that describes the input data format and state model that describes specific test input generation strategies to guide the fuzzing. The limitation of black-box fuzzing techniques is that the test input generation process lacks feedback from the program execution trace and results in low code coverage.

The greybox fuzzing techniques collect code coverage information, which directly affects the seed set involved in the mutation process. AFL [5] is the representative of many such techniques, which continually generate test inputs through deterministic and random mutations. AFLFast [6] optimizes AFL's strategy of selecting the next seed for mutation, where it preferentially mutates the seeds covering the low-frequency path. Similar to AFLFast, AFLGo [30] implements a simulated annealing-based power schedule, which helps to fuzz some target areas in the target binary. FairFuzz [10] modifies the mutation algorithm of AFL. It purposefully mutates specific bytes of selected seeds in both the deterministic and random mutation stages. PAFL [31] utilizes efficient guiding information synchronization and task division to extend existing fuzzing optimizations [6], [10] of single mode to industrial parallel mode. Combining fuzzing techniques with symbolic execution or program analysis [24], [32], [33] is another improvement in the field of greybox binary testing. For example, Driller [24] uses fuzzing to explore program paths under test most of the time. However, when the fuzzing becomes stuck, it uses concolic execution to generate new inputs for conditions that the fuzzer can not satisfy. However, these greybox fuzzing techniques are applicable to desktop binaries with a specific hardware architecture, and their usability and efficiency in embedded firmware are greatly reduced without the instrumentation to support the memory checker for vulnerabilities that would not result in crash.

B. Binary Memory Checking

Memory vulnerability checking is an important means of ensuring binary security. Common binary memory checking tools rely on different stages (e.g., compile-time, link-time, and runtime) of instrumentation to insert detection logic. Purify [34] is one of the first commercial memory inspection tools, relying on link-time instrumentation to detect use after free and memory leak vulnerabilities. MemCheck [7] is the most widely used tool for checking memory vulnerabilities. It implements memory checking on the Valgrind [35] dynamic instrumentation platform by first converting binary instructions into the VEX [35] intermediate representation (VEX-IR), then inserting detection algorithms on the VEX-IR, and finally converting them back to the binary instructions. Based on the DynamoRIO [22] instrumentation framework, Dr. memory [8] uses the code cache mechanism to support heap-based memory checking for binaries running on multiple operating systems. Parallel Inspector [36] is a memory and threading error debugger built on the Pin [37] dynamic instrumentation framework for the IA-32, x86-64 and MIC instruction set architectures. The above tools are workable on the premise that they can be installed in the same running environment (e.g., same CPU architecture) as the binary under test. Furthermore, those tools depend on extra test cases to trigger the execution and analysis procedure.

VII. Conclusion

In this paper, we present *EM-Fuzz*, an augmented fuzzing technique via memory checking to enhance vulnerability discovery capabilities on embedded firmware. Based on the dynamic instrumentation, we can insert multiple vulnerability checkers to detect those vulnerabilities that would not crash the system, and collect the sensitive memory operations to guard the fuzzer to generate more high-quality test cases and perform exhaustive memory checking on those fragile code regions. The experimental results on real-world embedded firmware manifest *EM-Fuzz* 's excellence not only in the number of vulnerability discovery but also in the speed boost. It has exposed 23 security vulnerabilities, 16 of which are newly discovered previously unknown vulnerabilities. In the future, we plan to extend *EM-Fuzz* by supporting more architectures and integrating more checkers for memory security threats.

References

[1] A. Cui and S. J. Stolfo, "A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 97–106.
[2] "Peach fuzzer," https://www.peach.tech/, accessed January 20, 2020.
[3] Y. Zhang, W. Huo, K. Jian, J. Shi, H. Lu, L. Liu, C. Wang, D. Sun, C. Zhang, and B. Liu, "Srfuzzer: an automatic fuzzing framework for physical soho router devices to discover multi-type vulnerabilities," in *Proceedings of the 35th Annual Computer Security Applications Conference*. ACM, 2019, pp. 544–556.
[4] 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 *NDSS*, 2018.
[5] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl, accessed January 20, 2020.
[6] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Trans. Software Eng.*, vol. 45, pp. 489–506, 2017.
[7] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *USENIX Annual Technical Conference, General Track*, 2005.
[8] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," *International Symposium on Code Generation and Optimization (CGO 2011)*, pp. 213–223, 2011.
[9] glibc wiki, "Introduction to malloc chunk," https://sourceware.org/glibc/wiki/MallocInternals, accessed January 22, 2020.
[10] C. Lemieux and K. Sen, "Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage," in *ASE*, 2018.
[11] QEMU, "The fast processor emulator," https://www.qemu.org, accessed January 22, 2020.
[12] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *USENIX Security Symposium*, 2018.
[13] ICCP, "Inter-control center communications protocol," https://en.wikipedia.org/wiki/IEC_60870-6#Inter-Control_Center_Communications_Protocol, accessed January 22, 2020.
[14] IEC104, "One of the iec60870 standard sets for supervisory control and data acquisition in electrical engineering and power system automation applications," https://en.wikipedia.org/wiki/IEC_60870-5, accessed January 22, 2020.
[15] IEC61850, "International communication protocol standard for intelligent electronic devices," https://en.wikipedia.org/wiki/IEC_61850, accessed January 22, 2020.
[16] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," *Commun. ACM*, vol. 59, pp. 93–100, 2014.
[17] CVE, "Common vulnerabilities and exposures," https://www.cvedetails.com, accessed January 22, 2020.

[18] Coreutils, "Gnu core utilities," https://github.com/coreutils/coreutils, accessed January 22, 2020.
[19] glennrp, "Libpng: Portable network graphics support," https://github.com/glennrp/libpng, accessed January 22, 2020.
[20] zlib, "A massively spiffy yet delicately unobtrusive compression library," http://zlib.net/, accessed January 22, 2020.
[21] jbeder, "A yaml parser and emitter," https://github.com/jbeder/yaml-cpp, accessed January 22, 2020.
[22] D. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," 2004.
[23] "Common vulnerability scoring system," https://en.wikipedia.org/wiki/Common_Vulnerability_Scoring_System, accessed January 20, 2020.
[24] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 2016.
[25] zzuf, "multi-purpose fuzzer," http://caca.zoy.org/wiki/zzuf, accessed January 20, 2020.
[26] K. C. Sang, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Security & Privacy*, 2015.
[27] Google, "honggfuzz," https://google.github.io/honggfuzz/, accessed January 20, 2019.
[28] libFuzzer, "a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, accessed January 20, 2020.
[29] J. Liang, Y. Jiang, M. Wang, X. Jiao, Y. Chen, H. Song, and K.-K. R. Choo, "Deepfuzzer: Accelerated deep greybox fuzzing," *IEEE Transactions on Dependable and Secure Computing*, 2019.
[30] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *CCS '17*, 2017.
[31] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, "Pafl: extend fuzzing optimizations of single mode to industrial parallel mode," in *ESEC/FSE*, 2018, pp. 809–814.
[32] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym : A practical concolic execution engine tailored for hybrid fuzzing," in *USENIX Security Symposium*, 2018.
[33] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *ICSE: Companion Proceeedings*, 2018, pp. 61–64.
[34] R. O. Hastings and B. Joyce, "Purify: fast detection of memory leaks and access errors," 1991.
[35] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007.
[36] Intel, "Intel parallel inspector," https://software.intel.com/en-us/inspector, accessed January 20, 2020.
[37] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.