What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices

Marius Muench*, Jan Stijohann†‡, Frank Kargl‡, Aurélien Francillon* and Davide Balzarotti*
*EURECOM. {muench, francill, balzarot}@eurecom.fr
†Siemens AG. jan.stijohann@siemens.com
‡Ulm University. frank.kargl@uni-ulm.de

Abstract—**As networked embedded systems are becoming more ubiquitous, their security is becoming critical to our daily life. While manual or automated large scale analysis of those systems regularly uncover new vulnerabilities, the way those systems are analyzed follows often the same approaches used on desktop systems. More specifically, traditional testing approaches relies on observable crashes of a program, and binary instrumentation techniques are used to improve the detection of those faulty states.**

In this paper, we demonstrate that memory corruptions, a common class of security vulnerabilities, often result in different behavior on embedded devices than on desktop systems. In particular, on embedded devices, effects of memory corruption are often less visible. This reduces significantly the effectiveness of traditional dynamic testing techniques in general, and fuzzing in particular.

Additionally, we analyze those differences in several categories of embedded devices and show the resulting impact on firmware analysis. We further describe and evaluate relatively simple heuristics which can be applied at run time (on an execution trace or in an emulator), during the analysis of an embedded device to detect previously undetected memory corruptions.

I. Introduction

Networked embedded systems are becoming a key component of modern life and the interconnection among different devices and their associated online services is at the core of the so called "Internet of Things". While it is impossible to accurately determine the precise number of such devices currently deployed worldwide, it is commonly agreed that this number is continuously growing and likely to exceed 20 billion by 2020 [38].

The rapid growth in connected devices is accompanied by an increase of their attack surface. This is especially daunting as those devices' applications not only span consumer electronics but also encompasses the medical sector, autonomous vehicles, Industrial Control Systems (ICS) and more. Although the importance of the security of those devices is vital, recent attacks are drawing a devastating picture of reality: Malware like Mirai was attributed to puppeteer more than tens of millions devices in 2016 [30], subsystems in vehicles of notable manufactures [26], [34] as well as third party devices [16] have been proven to be vulnerable to remote attacks, and more than 1200 attacks on ICS have been observed in 2015 alone [32].

A large number of vulnerabilities found today on embedded devices can still be considered as "low-hanging fruits", such as authentication or insecure management interfaces which could be mitigated by enhancing the awareness of both vendors and end-users. Another prevalent class of vulnerabilities that affect these devices, however, is due to programming errors leading to memory corruptions [39]. One of the most popular techniques to discover these flaws is fuzz-testing, which, unlike source code analysis and reverse engineering, is well suited for large-scale automation.

In a nutshell, fuzz-testing, or "fuzzing", describes the process of automatically generating and sending malformed input to the software under test, while monitoring its behavior for anomalies [50]. Anomalies themselves are visible ramifications of *fault states*, often resulting in *crashes*. Over the past decade, fuzzing became a major part of software development testing as well as a very common tool for third party software security testing.

Unfortunately, while common desktop systems have a variety of mechanisms to detect faulty states (e.g., segmentation faults, heap hardening and sanitizers) and to analyze them (e.g., command-line return values or core dumps), embedded devices often lack such mechanisms because of their limited I/O capabilities, constrained cost, and limited computing power. As a result, *silent memory corruptions* occur more frequently on embedded devices than on traditional computer systems, creating a significant challenge for conducting fuzzing sessions on embedded systems software.

Because of the incredible importance that those systems are taking, fuzzing needs to be applied to embedded devices as it is today on software running on desktop computers. It is therefore crucial to understand the difficulties in doing so and to develop alternative ways to detect and analyze memory corruptions.

Fuzzing can be performed with or without the availability of the source code. Source code availability makes testing more efficient as memory semantics can be used to detect anomalies, for example, by using compile time corruption detection techniques. However, fuzzing embedded devices –

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23166
www.ndss-symposium.org

which lack memory protections, exploit countermeasures, and for which source code is very rarely available – becomes quite difficult. Indeed, the only way left to identify a successful memory corruption is to monitor the device to detect signs of an incorrect behavior. In this paper, we show that these "liveness" checks are insufficient to detect many classes of vulnerabilities because it is often difficult to detect in a black-box manner when the internal memory of the embedded device has been corrupted.

While the aforementioned problems certainly have influenced previous fuzzing experiments on embedded devices conducted by other researchers, up to our knowledge, they have not been independently studied to this date. In this paper we therefore make the first comprehensive analysis of the effects of memory corruptions on different categories of embedded systems. The study would not be complete without discussing possible solutions that can be adopted to mitigate this problem. In particular, we investigate a combination of (partial or full) emulation with a number of runtime analysis techniques designed to replace the fault detection mechanisms present in traditional desktop computer systems. Our experiments show that the analysis techniques we propose can detect 100% of the corrupted states triggered during our fuzzing session, and therefore can significantly improve the number of detected vulnerabilities. We also show that partial emulation significantly reduces the fuzzing throughput. However, when full emulation is possible, this setup can combine the best of both worlds and detect 100% of the corrupting inputs while improving the fuzzing efficiency beyond what could be obtained using the physical device.

In summary, in this paper we make the following contributions:

- we analyze the challenges of fuzzing embedded devices,
- we make a classification of embedded systems with respect to the difficulty of detecting memory corruptions in their software,
- we evaluate the real world effects of different memory corruptions on those classes of embedded systems,
- we describe the techniques that can be used for improving fuzzing on embedded devices,
- we present six heuristics that can be used to detect faults due to the memory corruption, when the firmware of an embedded system can be run in either a partial or full emulation environment,
- we implemented our heuristics on top of a combination of the Avatar [54] and PANDA [13] frameworks and we conducted a number of tests to show their effectiveness and their overhead in a fuzzing experiment.

II. Fuzzing Embedded Systems

The term fuzz-testing, or simply "fuzzing", was first introduced in 1990 by Miller et al. [33] to describe an automated program testing technique in which the system under test was fed with random input data. Nowadays, fuzzing is one of the most prevalent techniques for vulnerability discovery and more sophisticated, or "smart", fuzzing techniques exist to guide the test case generation process. In general, modern fuzzers can be categorized in mutation-based fuzzers [42], [55], generation-based fuzzers [4], [15] and guided fuzzers [17]–[20], [49], based on the input generation strategy.

Most of the theory behind fuzzing and most of the available fuzzing tools were designed to test software running on desktop PCs. As we will discuss later in this paper, there are a number of relevant differences that makes fuzzing embedded system particularly challenging. As those differences depends on the characteristic of the device under test, we first introduce the classification of embedded systems that we will use in the rest of the paper. We then discuss previous experiments that applied fuzz testing to different embedded devices, and finally we present the challenges encountered to apply fuzzing in this domain.

A. Classes of Embedded Devices

While a precise definition of what is an embedded system is hard to establish [21], in this paper, we adopt the widely accepted idea that embedded devices are separated from modern general-purpose computers by two common characteristics: a) embedded systems are designed to fulfill a *special purpose*, and b) embedded systems often interact with the physical world through a number of *peripherals*[1] connected to sensors and actuators. The aforementioned two criteria cover a wide variety of devices, ranging from hard disk controllers to smart thermostats, from digital cameras to PLCs. These families can be further classified according to several aspects, such as their actual computing power, their unit cost, the field of usage, the extent to which they interacts with the environment, or the timing constraints imposed on the device.

Yet, these classifications tell very little about the type of security mechanisms that are available on a given device. Therefore, in this paper we divide embedded systems according to the type of operating system (OS) they use. While the operating system is certainly not the only source of security features, it is responsible for handling the recovery from faulty states, and it often serves as building block for additional, more complex, security primitives.

We separate embedded devices in the following three classes:

Type-I: General purpose OS-based devices.
General purpose Operating Systems are often retrofitted to suit embedded systems. However, in comparison to the traditional desktop or server counterparts, embedded systems typically follow more minimalistic approaches. For example, the Linux OS kernel is widely used in the embedded world, where it is typically coupled with lightweight user space environments (e.g., `busybox` and `uClibc`).

Type-II: Embedded OS-based devices.
In recent years, custom operating systems for embedded devices have gained popularity. These systems are particularly suitable for devices with low computational power, and while advanced processor features such as a Memory Management Unit (MMU) may not be present, a logical seperation between kernel and application code is still present. Operating systems such as `uClinux`,

[1] Note that while embedded systems can either be self-contained or consist of several *embedded devices*, for the sake of simplicity, we will use the term embedded systems analogously to embedded device, as our work is focused on single devices, rather than on interconnected systems.

Study	Year	Sector	Fuzzing Approach	Type of Device		
				Type-I	Type-II	Type-III
Koscher et al. [27]	2010	Automotive	Mutation-based, Blackbox	-	-	✓
Mulliner et al. [35]	2011	GSM feature phones	Generation-based, Blackbox	-	✓	-
Kamel et Lanet [23]	2013	SmartCards	Generation-based, Blackbox	-	-	✓
Almgren et al. [3]	2013	PLCs & SmartMeters	Mut.- & gen.-based, Blackbox	-	✓	-
Alimi et al. [2]	2014	SmartCards	Generation-based, Blackbox	-	-	✓
Van den Broek et al. [51]	2014	Smartphones	Generation-based, Blackbox	✓	✓	-
Kammerstetter al. [25]	2014	–	Mutation-based, Dynamic Inst.	✓	-	-
Lee et al. [29]	2015	Automotive	Random-based, Blackbox	-	-	✓

TABLE I: Fuzzing experiments of embedded systems in the literature.

`ZephyrOS` or `VxWorks` are examples for these systems and they are usually adopted on single-purpose user electronics, such as IP cameras or DVD players.

Type-III: Devices without an OS-Abstraction.
These devices adopt a so called "monolithic firmware", whose operation is typically based on a single control loop and interrupts triggered from the peripherals in order to handle events from the outer world. Monolithic approaches can be found in a large variety of controllers of hardware components, like CD reader, WiFi cards or GPS dongles. The code running on these devices can be completely custom, or it can be based on *operating system libraries* such as `Contiki`, `TinyOS` or `mbed OS 2`[2]. While those libraries are providing abstractions to programmers, the resulting firmware will contain both system and application code compiled together, and, thus, forms a monolithic software.

For better readability, we will use Type-I, Type-II, and Type-III inter-exchangeably with the corresponding class names. Additionally, we will use Type-0 in order to reference to traditional multi-purpose systems.

B. Past Experiments

In recent years, fuzz-testing has become more and more popular as a way to test the security of embedded systems. Table I provides a short overview of some of the recent efforts in this area. These works covered different sectors, different input generation strategies, and different classes of embedded devices.

For instance, Alimi et al. [2] fuzzed parts of the MasterCard M/Chip specifications by generating test-inputs using a genetic algorithm. The authors observed that real cards would become unusable during comprehensive fuzz-testing, and therefore moved the parts under test into a simulator where they could trigger the approval of illegitimate transactions.

Some modern smart cards also contain a web server implementation. Kamel et Lanet [23] designed a generation-based fuzzer for the HTTP implementations of those web servers and were able to trigger several flaws, including errors in the smart cards. Koscher et al. [27] conducted a security test of automotive systems using fuzzing in addition to reverse engineering and packet sniffing. More specifically, the authors fuzzed packets for the CAN bus, discovering packets to unlock all doors, disable the car's light, and permanently enable the horn. Similar results were reported in a later study by Lee et al. [29]. Hereby, random fuzzing of the data field in CAN packets led at least to observable changes in the car's instrumentation panel.

In [3], Almgren et al. developed several mutation-based and generational-based fuzzers which were used to test various PLCs and smart meters. The fuzzing experiments lead to the discovery of several known and unknown denial of service vulnerabilities, some leading to a completely unresponsive PLC which could only be recovered after a power cycle and a cold restart.

Mulliner et al. [35] and Van den Broek et al. [51] developed a generation-based fuzzer in accordance to the GSM-specification to test GSM implementations in both feature- and smart-phones. While both studies were able to trigger a large number of errors – including memory exhaustions, reboots, and denial-of-service conditions – the authors concluded that correctly monitoring the devices under test in an automated manner is still a very challenging task.

To avoid this problem, PROSPECT [25] introduces a novel approach that involves the partial emulation of an embedded device's firmware during a fuzzing experiment. The authors were able to emulate Linux-based systems by forwarding system calls that are likely to access peripherals to the physical device. Using this method, the authors discovered a previous unknown 0-day vulnerability in a fire alarm system.

C. Main Challenges of Fuzzing Embedded Devices

Based on our experience and on problems reported by other authors while conducting fuzzing experiments in previous works, we identified three main challenges that make fuzzing much more complex and less efficient on embedded devices compared to desktop systems. On top of these specific problems, embedded systems are also inheriting other challenges which are common to fuzzing in general (e.g., test case generation) but we consider them out the scope of this work.

[C1] – Fault Detection. The vast majority of fuzzing techniques relies on *observable crashes* as immediate consequences of faults occurring during a program's execution. Desktop systems offer a variety of protection measurements which are triggering a crash upon a fault, and while some of them are designed with security in mind (e.g., stack canaries), others are inherent architecture artifacts, such as segmentation faults caused by a Memory Management Unit (MMU). Unfortunately, these techniques are rarely present (or are very

[2] Interestingly, and despite the name, mbed OS 2 (also referenced as "mbed OS classic") is an operating system library. Later versions, on the other hand, are actual embedded OS and would serve as building block for Type-II devices.

limited) on embedded devices. As we will describe in more details in Section III, this poses a significant challenge for fuzzing embedded devices.

Moreover, even when mechanisms leading to observable crashes are in place, monitoring those crashes can be complicated. In fact, while for desktop systems crashes are often accompanied with error messages, embedded systems may lack equivalent I/O functionalities.

As a result, a sophisticated *liveness* check (or *probing*) has to be deployed to check the effects of the test on the device. In general, it is possible to adopt two types of probing. *Active probing* inserts special requests into the communication to the device or to its environment. This influences the state of the software under test – as the state of the software is periodically checked by providing *valid input* and evaluating the corresponding response. In contrary, *passive probing* aims at retrieving information about the device's state without altering it. This could be achieved by looking at the answers provided by the device to the test inputs or by observing "visible" crashes.

[C2] – Performance and Scalability. In case of desktop systems, multiple instances of the same software can easily be started and fuzzed in parallel, e.g., via multi-processing or virtualization. This parallelization poses a substantial challenge when dealing with embedded devices. Obtaining a large number of copies of the same physical device is often infeasible due to limited resources (e.g., financial), and to infrastructure requirements such as space and power supply.

Additionally, fuzzing frequently requires to restart the target under test in order to re-establish a clean state for the next test case. While this is easy to achieve for desktop systems, e.g., via virtual machine snapshots, resetting the state of an embedded system can take a considerable amount of time (up to few minutes), as it often requires a full reboot of the device. This results in a considerable slow down of the entire testing process, as we will discuss in more details in Section VI.

[C3] – Instrumentation. While fuzzing was originally proposed as a black-box testing technique, it quickly became evident that a more clever generation and prioritization of the input values required access to some information about the state of the system under test. For example, it is very popular for desktop systems to adopt a mix of *compile-time instrumentation* and *run-time instrumentation* to collect code-coverage information about the provided inputs and to detect subtle corruption of the memory of the system.

Probably the most established fuzzer which uses source code instrumentation to collect information about retrieved code coverage is American Fuzzing Lop (AFL) [55]. Prominent examples of instrumentation to detect memory corruptions are sanitizers like AddressSanitizer [43], ThreadSanitizer [44], MemorySantizer [48], and UndefinedBehaviourSanitizer [11], which also have proven to be very efficient in uncovering vulnerabilities in combination with fuzzing [45].

While all of these components are available for the popular compilers `clang` and `gcc`, source code is not typically available for firmware images from embedded devices, and in many cases its is challenging to re-compile the software. In fact, embedded systems can span a variety of devices, each with its own operating system, peripherals and processor – for which a comprehensive toolchain is rarely available to the tester. A common solution that does not require access to the source code of the application is to resort to binary dynamic instrumentation frameworks such as Pin [31], Valgrind [37], and DynamoRio [7]. AFL is also capable of retrieving coverage-information while fuzzing binary software by adding instrumentation capabilities to QEMU, which has recently even be leveraged for fuzzing full-blown Linux kernels [36].

Unfortunately, all of the above static and dynamic instrumentation tools are closely tied to the target operating system and CPU architecture, and at the time of writing none of them provides support for Type-II and Type-III embedded devices.

III. Memory Corruptions in Embedded Systems

In this section we investigate the effects on different classes of embedded devices of memory corruption vulnerabilities, a class of bugs which often leads to crashes on desktop systems. For better readability, we first introduce a definition for *observable crash* and *fault states*. Afterwards we will describe our case study and the obtained results.

A. Bugs, Faults, Corruptions & Crashes

In general, both firmware and software are built to solve specific tasks, and thus they follow a – not necessarily explicit – specification of what they are intended to do. It is also well known that software and firmware code can contain bugs: errors that can bring a program into an unintended state. When such unintended state can be exploited by an attacker, a bug is classified as a security vulnerability.

A common class of bugs that often leads to vulnerabilities are memory corruptions or memory errors. In particular, Van der Veen et al. [52] differentiate between spatial and temporal memory errors. Whereas the first type denotes out of bounds accesses of a memory object, the second type represents accesses to a memory object that does not exist anymore.

A memory corruption itself can cause an *observable crash* of a program, whereby the program is either terminated or some recovery procedures, such as exception handlers, are executed. Today, many common security mechanisms – such as stack canaries or heap consistency checks – trigger these crashes. However, under particular conditions, we can encounter memory corruptions which do not lead to any observable and immediate crash of the system. We call these cases *silent memory corruptions*. In a silent memory corruption the program continues its execution and enters an unintended *faulty state*.

The important consequence of silent memory corruptions is that the actual fault might only become noticeable at a later point in time when a certain functionality is requested or when a particular sequence of events is received. While this may not be considered a problem as long as the device continues to executes, it poses a significant threat for the safety and security of embedded devices. In fact, as soon as the system enters a faulty state, the integrity of its operation cannot be guaranteed anymore, and wrong data could be returned or processed at any time.

	Platform	Manufacturer & Model	CPU Family	Operaring System	LIBC	MMU
Desktop	Single Board Computer	Beaglebone Black	Cortex A-9	Debian GNU/Linux	glibc	✓
Type-I	Router	Linksys EA6300v1	Cortex A-9	Embedded Linux	uclibc	✓
Type-II	IP camera	Foscam FI8918W	ARM7TDMI-S	uCLinux	uclibc	✗
Type-III	Development Board	STM Nucleo-L152RE	Cortex M-3	None	libmbed	✗[3]

TABLE II: Devices selected for the experiments.

While desktop systems are also subject to silent corruptions, those are a lot less frequent because they are deploying several lines of defenses. Indeed while those lines of defense are often in fact designed for hardening programs against attacks they also make faults more likely to lead to a crash. Those mechanisms include memory isolation, protection mechanisms, and memory structures integrity checks.

Embedded systems, on the other hand, often lack similar mechanisms. This can not only lead to devastating consequences as soon as a system interacts with the exterior world, but it also complicates the security analysis of those systems as many black box testing techniques, and fuzzing in particular, rely on observable effects to infer the state of the device.

B. Experimental Setup

In order to study how memory corruptions behave across different computing systems, we conducted a number of experiments. Our goal is to trigger the same memory corruption conditions on different systems to analyze whether they yield to observable crashes or result in silent corruptions.

For these experiments, we selected one device for each device class presented in Section II-A, and compare the results with a baseline system consisting of full-fledged GNU-Linux desktop OS. All systems are ARM-based and we recompiled each firmware image to obtain comparable results. We introduced our vulnerabilities in two popular and widely used libraries: *mbed TLS*, an SSL library designed for both embedded and desktop system, and *expat*, a popular XML parser. The two are good candidates for our experiments because memory corruptions vulnerabilities were previously publicly released in both of them, and because they are popular, open source, and present in many modern embedded devices.

To analyze their behavior in a realistic context, we chose existing Commercial Off-The-Shelf (COTS) products: a router to represent Type-I systems, and an IP camera for Type-II devices. We compiled our vulnerable application for those targets and then loaded it on the device. For the monolithic class, however, obtaining a COTS device with customizable firmware is difficult, as their firmware usually consists of only one custom binary blob responsible for the entire operation of the device. Therefore, we used a development board with publicly available software for peripheral interaction. For this device we included our test code in the firmware, compiled it and then loaded the firmware on the device. Table II shows a summary of our three test platforms, including information about which C library is used and whether an MMU is present. Besides the operating system, these properties mainly determine the behavior of a system in case of a memory corruption.

Listing 1: Examples of artificial vulnerabilities.

```
XML_Parse(XML_Parser parser, const char *s, int len,
     int isFinal)
{
  char overflowable[128];
  [...]
  //this returns a heap object
  void *buff = XML_GetBuffer(parser, len);
  [...]
  //trigger immediate stack-based buffer overflow
  if (len == 1222){
    memcpy(overflowable, s, len);
    return;
  }
  //this will cause a null pointer dereference
  else if (len == 1223){
    buff = NULL;
  }
  //causing a heap-based buffer overflow
  if (len == 1225){
      memcpy(buff, s, len);
      memcpy(buff + 1225, s, len);
  }
  else{
    memcpy(buff, s, len);
  }
  //cause an uncontrolled format-string
    vulnerability
  if (len == 1224){
    printf(buff);
  }

  [...]
}
```

C. Artificial Vulnerabilities

Since our focus is not the discovery of new bugs but rather the analysis of the effects of memory corruptions on embedded systems, we inserted several vulnerabilities leading to memory corruptions in our test samples. Specifically, we used stack-based buffer overflows and heap-based buffer overflows as examples of spatial memory corruptions, and null pointer dereferences and double free vulnerabilities as examples of temporal memory corruptions. Additionally, we also inserted a format string vulnerability that can either be used for information leakage or for arbitrary memory corruptions.

Our approach of bug insertion was inspired by the one used in LAVA [14], i.e., we ensured that each vulnerability had its own independent trigger condition. However, as we only had to inject a limited number of vulnerabilities, we *manually* selected the vulnerable paths and the position of each bug. Likewise, for the purpose of our experiments we did not need “realistic” checks or path conditions particularly difficult to

[3] Note that Cortex M-3 microcontrollers can be equipped with an optional Memory Protection Unit (MPU), which provides basic memory protections. While present on this specific Type-III device, we do not utilize its features, as this is a very common scenario and the most problematic case.

	expat				**mbed TLS**			
Platform	**Desktop**	**Type-I**	**Type-II**	**Type-III**	**Desktop**	**Type-I**	**Type-II**	**Type-III**
Format String	✓	✓	✗	✗	✓	✓	✗ (malfunc.)	! (hang)
Stack-based buffer overflow	✓	✓	✓ (opaque)	! (hang)	✓	✓	✓ (opaque)	! (hang)
Heap-based buffer overflow	✓	! (late crash)	✗	✗	✓	! (late crash)	✗	✗
Double Free	✓	✓	✗	✗ (malfunc.)	✓	! (late crash)	✗	✗
Null Pointer Dereference	✓	✓	✓ (reboot)	✗ (malfunc.)	✓	✓	✓ (reboot)	✗

TABLE III: Observed Behaviour of triggered memory corruptions.

explore. Therefore, we simply added custom branches in the code that triggered the various vulnerabilities based on the length of the user-provided payload. Listing 1 shows four of the inserted vulnerabilities inside the expat library.

D. Observed Behavior

The goal of this experiment was to observe the behavior of each device after sending malicious inputs that trigger one of the inserted vulnerabilities. As expected, the software running on GNU/Linux desktop crashed each time it was provided with a malicious input that triggered the vulnerability.

However, the embedded devices were not always able to detect the fault and, in some cases, they even continued the execution with no visible effects – despite the fact that the underlying memory of the system was corrupted. To better differentiate between the different behaviors we observed in our experiments, we categorize our observations in six possible results:

[R1] **Observable Crash** (✓) – The execution of the device under test stops and a message or other visible effect is easily observable. In less optimal cases, no detailed information about the causes of the crash is produced (we mark these cases as `opaque` in Table III).

[R2] **Reboot** (✓) – The device immediately reboots. For Type-III devices there is no difference between a crash and a reboot because they are monolithic applications. However, for Type-I and Type-II devices a given service (e.g., a web server) can crash while the rest of the embedded system may still continue to work properly.

[R3] **Hang** (!) – The target hangs and stops responding to new requests, possibly stuck in an infinite loop.

[R4] **Late Crash** (!) – The target system continues its execution for a non-negligible amount of time and crashes afterwards (e.g., when the connection is terminated).

[R5] **Malfunctioning** (✗) – The process continues, but reports wrong data and incorrect results.

[R6] **No Effect** (✗) – Despite the corrupted memory, the target continues normally with no observable side-effects.

Immediately observable crashes and reboots (R1 and R2) are the preferred outcomes of an experiment as during a fuzzing session they allow to immediately identify the responsible input.

Hangs and late crashes (R3 and R4) can be more difficult to deal with, in particular when the crash is delayed long enough that a fuzzer may have already sent multiple other inputs to the target system and the input responsible for the corruption will therefore be difficult to identify. However, the presence of a fault is still observable in these cases. Things get even more complex when a device starts malfunctioning (R5). In this case, there are no crashes at all, but the results provided to certain requests may be incorrect. To detect this case, a fuzzer would need to know what is the correct output for each input it sends to the system – something which is very seldom the case in security testing. A possible workaround can consist in inserting between two consecutive inputs a number of functional test requests for which the output is known. However, even when this solution is sufficient to detect the malfunction, it introduces a considerable delay in the fuzzing experiment. Finally, the worst case is when the device continues its operation with no observable side effect (R6). In fact, since part of the device memory has been corrupted, there may be side-effects or unexpected behaviors in the future.

Table III shows the result of our experiments. It is clear that the fewer features are provided by an embedded platform, the less likely the system is to detect memory corruptions. An interesting observation is already the difference between a full scale GNU/Linux and an embedded Linux for heap-based buffer overflows and double free corruptions. While on the desktop system the inlined heap consistency checks provided by the standard C library are triggering a verbose crash quickly after the corruption, the embedded Linux continues and crashes at a later point in time, often just during the `exit()` handler.

The table also shows that corrupting inputs for Type-II and Type-III devices are very rarely triggering a crash. This provides a perfect example of how common are *silent* memory corruptions in real-world embedded systems.

Another important observation can be made when looking at the devices' behavior for the format string vulnerability. Both the embedded and the full scale GNU/Linux are reporting segmentation faults, due to the attempt to access unmapped memory. However, uClinux and the monolithic device are continuing execution, which is a result of the lack of an MMU. This shows that the MMU plays a very important role when it comes to detecting memory corruptions.

A similar behavior can be observed on the results of the null pointer dereference test. The processes running on both the GNU/Linux desktop and on the embedded Linux are correctly crashing once the program tries to write memory to the NULL-address range. With the same vulnerability, the monolithic device will continue execution although data has been written to the exact same address. In fact, as the execution of the firmware is not dependent on the content of the memory at this location, this memory corruption does not influence the behavior of the system.

The result of the same test on the uClinux system is particularly interesting: after the NULL write, the device hangs for few seconds and then reboots. This is not surprising as in uClinux the kernel is mapped in the lower part of the memory (so writing at address 0x0 corrupts kernel memory). The reboot, on the other hand, is possibly caused due to an hardware watchdog that detects the hang and automatically restarts the device as a recovery mechanism.

To summarize, while in certain conditions silent memory corruptions can occur also in traditional desktop envoronments, our experiments show that they are often the rule and not the exception in the embedded world. As the use of fuzzing is becoming prevalent to test firmware code for vulnerabilities, and as fuzzing is relying on observable crashes to detect bugs, the limited support for fault detection can have very severe consequences on the effectiveness of security testing for embedded devices.

IV. Mitigations

In the previous section we have seen that embedded systems come in all forms and with very different characteristics. While fuzzing a Type-I device may not present particular challenges, memory corruptions on Type-III systems rarely result in an immediate crash, imposing a significant challenge to automatically identify when a vulnerability has been triggered. So how can we fuzz devices when no reliable feedback is available? Again, the vast diversity of existing devices makes it difficult to find a general answer to this question. Therefore, in this section we present six different options that may be available to the tester and we discuss both advantages and limitations of each solution.

A) Static Instrumentation.

Prerequisites: source code and compiler toolchain

As the majority of software for embedded devices is distributed in a binary-only manner, access to the source code can be considered an exceptional case. In the rare cases in which the source code is available, proprietary toolchains are often required to recompile the firmware.

However, when testers have all the required components (e.g., if the tester is the device manufacturer) they can instrument the source code to provide better run-time information or to introduce a crash harness functionality. Typical instrumentation includes a combination of multiple techniques [45]. For example, collecting a trace of execution to measure coverage to mutate fuzzing inputs [55], adding checks for memory allocations [43], or hardening the program with, e.g., Control Flow Integrity [1]. Unfortunately, most of these tools are not yet available for embedded systems.

B) Binary Rewriting.

Prerequisites: binary firmware image, device

When only the binary firmware is available, binary rewriting can be used to instrument the code [**?**]. However, there are some difficulties in instrumenting the code. First, the firmware needs to be fully disassembled which can be difficult when it is provided as a raw binary [5]. Second, memory semantics, boundaries and data structures are lost in the compilation process and need to be recovered to add instrumentation. This requires a very challenging partial decompilation phase. Finally, the memory usage of embedded devices is often optimized to reduce costs, leaving little room for adding complex instrumentations.

C) Physical Re-Hosting.

Prerequisites: sources, compiler chain, different device

In certain cases, the analyst may be able to recompile the code for a different target device, for example to relocate a process from a Type-II device to a more test-friendly Type-I device or from a Type-I device to a regular Linux desktop system. This may also improve *scalability*, if the new device is cheaper and more readily available than the original one or if the new target is a regular computer.

However, on top of the difficult prerequisites, methods relying on this approach have another major drawback. In fact, it may be difficult to reproduce bugs found on the new target system in the original device (where they may even not be present at all due to the different architecture or due to changes introduced by recompiling the binary) and conversely, bugs that are present on the original target may not be present on the new target.

D) Full Emulation.

Prerequisites: firmware image, peripherals emulation

Images of the device firmware are often available to the analyst, either because they are extracted directly from the device or because they are obtained from a firmware update package available from the manufacturer. Costin et al. [12] and Chen et al. [8] have shown that, under certain conditions, applications extracted from Type-I firmwares can be *virtually rehosted*, i.e., they can be executed inside a generic operating system running on a default emulator. Likewise, the Qemu STM32 [6] project, which extends Qemu to emulate the STM32 chip, shows that when complete hardware documentation is available, with a considerable effort to implement the hardware emulation it is also possible to fully emulate Type-III firmware images.

This solution can greatly improve fuzzing. First of all, test experiments can be conducted *without* the presence of the physical device, thus allowing for a much greater parallelization. Second, dynamic instrumentation techniques can be easily applied and the emulator can be used to collect a large amount of information about the running firmware. The disadvantage of this solution is that it is only applicable when all peripherals being accessed by the target are known and can be successfully emulated, which is unfortunately rarely the case. Overall, being able to run an arbitrary firmware inside an emulator still remains an open research problem.

E) Partial Emulation.

Prerequisites: firmware image, device

If full emulation remains impractical in most circumstances, a partial emulation can still provide benefit while conducting testing experiments. This approach was first proposed by Avatar [54] and Surrogates [28] for Type-III devices and then extended to Type-I systems in PROSPECT [24], [25]. The general idea behind this solutions is to use an emulator (in which the firmware code is executed) modified to forward peripheral interactions to the actual physical device. The result provides the advantages of a full emulation solution without the burden of knowing and emulating I/O operations. However, what this solution gains in flexibility it is sacrificed in performances (due to the additional interaction with the real device) and scalability (due to the current need of pairing each emulated instance with a physical device).

F) Hardware-Supported Instrumentation.

Prerequisites: device, advanced debug feature (e.g., trace capable debugging port and debugger)

If the tester has access to a physical device with advanced hardware instrumentation mechanisms (such as real time tracing), it may be possible to collect enough information to improve the fault detection during the execution of the device. For instance, chip manufacturers often embed hardware tracing features such as ARM's Embedded Trace Macrocell (ETM) and Coresight Debug and Trace, or Intel's Processor Trace (PT) technologies [41] [4]. ARM tracing mechanisms are optional components of processors ("IPs"), which may be optionally included in the processor. Multiple variants tracing exist, such as tracing only branches, all instructions, or also all memory accesses – and different techniques rely on different debug ports (e.g., dedicated trace ports, *Single Wire Debug* (SWD) or *Single Wire Output* (SWO) ports). Unfortunately, the availability of such tracing hardware is variable. In lower-end devices (typically Type-III devices), manufacturers tend not to include any tracing capabilities, because of the relatively large impact on the chip surface, and therefore on the cost, that such mechanisms would incur. Development devices may have such facilities (sometimes when the micro-controller design is tested on FPGA before manufacturing) but this is less frequent in commercial production devices. Finally, in some cases debug access may be present but not available to prevent third-party analysis.

For example, while looking for test devices to conduct our experiments we encountered devices where the tracing support was deactivated for security reasons, multiplexed on pins which are used for another purpose, not routed on the circuit board (PCB), or where the silicon supported the feature but the pins were not available. In summary, when testing real world devices, the chances of finding an available and usable hardware tracing support are quite low.

	Execution	Register state	Memory Accesses	Memory Map	Annotated Program
Segment Tracking	✗	✗	✓	✓	✗
Format Specifier Tracking	✓	✓	✗	✓	✓
Heap Object Tracking	✓	✓	✓	✗	✓
Call Stack Tracking	✓	✗	✗	✗	✗
Call Frame Tracking	✓	✓	✓	✗	✗
Stack Object Tracking	✓	✓	✓	✗	✓

TABLE IV: Deployed live analysis techniques and their requirements.

To summarize, the first three solutions (**A**, **B**, and **C**) require the tester to modify the firmware image, either statically through recompilation or dynamically through runtime instrumentation. Unfortunately, as we already explained above, this is rarely an option when conducting a third party security testing. At the other end of the spectrum, the last three techniques (**D**, **E**, and **F**) have the advantage of not requiring any firmware modification, but they require instead additional technologies (either a software emulator or an hardware tracing support) to collect information about the running firmware.

In this paper we show that this information can be used to detect the presence of faults during the execution of an embedded system, by replacing the role of memory protection or other crash harness mechanisms usually provided by the hardware or by the operating system on desktop computers. However, this require the development of dedicated analysis routines specifically designed to identify the signs of memory corruption.

V. Fault Detection Heuristics

In this section we present a set of heuristics that we implemented to mimic existing compile-time, and run-time, techniques, and which are already able to detect all the memory corruption vulnerabilities showcased in Section III. These heuristics are inspired by techniques that have already been in use for detecting or mitigating memory corruptions in other settings, such as shadow stacks, compiler warnings for unsafe function calls, and run-time verifications as implemented by instrumentation tools like Address Sanitizer [43] or Valgrind [37][5].

It is also important to mention that we selected heuristics to be implementation independent, in order to not only work in a live analysis setting (as it is the case if the firmware is run in an emulator) but also to be applicable "post-mortem" on previously collected execution traces (as in the case of an

[4] However, only available on recent high performance processors.

[5] While the underlying algorithms of the heuristics are based on known principles, we implemented them as external monitors (cf. Section VI). The interested reader can find the implementations at `https://github.com/avatartwo/ndss18_wycinwyc`.

hardware-based tracing mechanism). As a result, they only rely on information extracted from the execution of the binary code and additional annotations provided by the analyst.

In the following we introduce briefly six heuristics and discuss their role, their limitations, and the information they require in order to be applied in our scenario.

Segment Tracking:
Segment tracking is possibly the simplest technique that aims to detect illegitimate memory accesses. The core idea is to observe all memory reads and writes and verify if they occur in valid locations, thus somehow imitating an MMU at detecting *segmentation faults*. This technique only requires knowledge about the memory accesses and the memory mappings of the target. Both are easily accessible in an emulator, however, when only traces are available the memory map can be obtained by reverse engineering.

Format Specifier Tracking:
Tracking format string specifiers is a naïve technique to discover insecure calls to `printf()`-style functions and is inspired by the printf protection outlined in [46]. In essence, this protection validates that the format string specifier points to a valid location upon entry in a function of the `printf()` family. In the simplest case, without presence of dynamic generated format string specifier, those valid locations would have to lie within read-only segments. All in all, this technique requires not only knowledge about the location of format handling functions, but also the register state while entering one of those functions and the argument order. Both the location of the according function and their argument order can be obtained by reverse engineering or automated static analysis of the firmware.

Heap Object Tracking:
This technique is designed to detect both temporal and spatial heap related bugs and is influenced by the instrumentation and run-time verification approaches presented in [43]. It achieves its goal by evaluating the arguments and return values of *allocation* and *deallocation* functions and bookkeeping the location and sizes of heap objects. This allows to easily detect out-of-bounds memory accesses or access to a freed object. However, this heuristic depends on a variety of information: executed instructions, the state of the registers, memory accesses, and knowledge about allocation and deallocation functions. The latter could be retrieved by reverse engineering, or by using advanced methods for discovering custom allocators as demonstrated by MemBrush [9].

Call Stack Tracking:
Call stack tracking is replicating traditional shadow stack protections [53], therefore aiming at detecting functions that do not return to the callee. This can help to identify stack-based memory corruptions that overwrite the return address of a function. It does so by monitoring all direct and indirect function calls and return instructions. However, as embedded devices are often interrupt-driven, this heuristic can lead to false negatives. Nevertheless, it is especially appealing as it requires the least amount of information: only the knowledge of the executed instructions.

Call Frame Tracking:
Call frame tracking is a more advanced version of the call stack tracking technique which detects coarse grained stack-based buffer overflows, without false negatives, right when they occur. In essence, stack frames are located by tracking function calls, then contiguous memory accesses are checked not to cross stack frames. Hereby, this requires to identify the executed instructions as well as register values to extract the stack pointer values upon function entries. Then, memory accesses have to be observed to detect the actual corruptions.

Stack Object Tracking:
Stack objects tracking consists in a fine-grained detection of out-of-bound accesses to *stack variables*, which is inspired by the heap object tracking approach proposed by Serebryany et al. [43]. Hereby, memory reads and writes observed during execution are checked against the individual variable size and position in the stack. Obviously, this requires to track executed instructions and memory accesses, as well as elaborate information about the stack variables. For the sake of simplicity, we use variables information which is present in debug symbols. However, in the general case, it is possible to retrieve such information in an automated manner from binary code, as proposed by several previous studies [22], [47].

Table IV lists the six presented heuristics and summarizes what type information is required for the analysis.

VI. Implementation

In the previous section we introduced six heuristics which can aid fuzzing when applied during emulation or when used to analyze execution traces. In order to show their effectiveness, we implemented them as part of distinct proof of concept trace analysis and live analysis systems. However, as the availability of hardware trace capabilities is uncommon and the live analysis scenario presents more engineering challenges, we focus our experiments on performing live detection while fuzzing a firmware that is either partially or fully emulated.

The experimental live analysis system we created combines PANDA [13], a dynamic analysis platform, and Avatar [54], an orchestration framework for dynamic analysis of embedded devices. Our implementation required modifications to both frameworks and we released both the modifications and the implemented heuristics as open source [6], to help future fuzzing campaigns and motivate additional research in this field.

Live analysis system description: The three core components of the experimental system are Avatar, PANDA, and a set of PANDA analysis plugins which implements the heuristics described above. Avatar allows dynamic binary analysis of embedded devices by enabling *partial emulation* of Type-III devices, as described in section IV. In essence, it orchestrates the execution of the firmware on a target embedded device and

[6] Available at https://github.com/avatartwo/ndss18_wycinwyc.

on an emulator, by providing the possibility to automatically transfer state between device and emulator and by forwarding I/O accesses. Avatar was originally designed to use S2E [10] as backend emulator. As part of our work, we modified it to replace S2E with PANDA, as this solution allows not only to perform analysis *during* the execution, but also to create lightweight records of the execution which can be used to identify the root cause of a crash.

Like S2E, PANDA is based on the full system emulator QEMU and its plugin systems allows to hook various events, such as physical accesses to memory, translation, and execution of translated blocks. Additionally, the base implementation of PANDA provides already several analysis plugins, whereas one of them is of special use for our heuristics: `callstack_instr`. This plugin allows to register further callbacks on function calls and returns, and provides information about the current call stack, which simplify the implementation of several of our heuristics. However, as this plugin would return wrong information when the state of the application is corrupted, we only use its `on_call` event to detect the beginning of a function, while information of function returns are retrieved by analyzing the executed blocks.

In summary, we use PANDA to emulate the firmware and rely on its plugin system to obtain live feedback over the execution of a partial or fully emulated firmware. All this while Avatar orchestrates the execution and selectively redirect execution and memory accesses to the physical device.

State caching: Avatar also plays another important role in our system, allowing the tester to save and replay the device state after it is initialized. In fact, since the initialization procedures of embedded systems usually set up all peripherals, this phase impose a significant overhead every time the fuzzer needs to restart the device. Moreover, these procedures involve a large amount of I/O operations that have a negative impact on the execution of a partial emulator (that needs to forward every access to the physical device). However, as long as the peripheral interaction only concerns *stateless* peripherals, this overhead can be removed by taking advantage of the ability of emulators to execute from an initial snapshot. Therefore, our system can benefit from the ability of Avatar to save a snapshot of the device state after the its initialization is completed and reuse it to initialize the emulator.

VII. Experiments

In this section we present a number of experiments we conducted to test our heuristics both in a *partial* and in a *full emulation* scenario. Our goal is to show that it is possible to integrate such heuristics in a live fuzzing experiment, thus providing fault detection to mitigate the lack of equivalent mechanisms in the embedded system's platform and operating system.

However, our approach may introduce a non negligible overhead on the performance of the system, effectively increasing the time required to perform the testing session. Therefore, we also decided to measure the effects of our solution on the fuzzing throughput under different setup configurations.

A. Target Setup

For our tests, we compiled the *expat* application presented in Section III-B for a Type-III device[7] and conducted a number of fuzzing sessions against the target using four different configurations, covering both optimal and worst case scenarios:

- **NAT: Native.** Fuzzing is performed directly against the actual device – therefore without using any fault detection heuristic. We use this case as the baseline to compare the results of other experiments.
- **PE/MF: Partial Emulation with Memory Forwarding.** The firmware is emulated and access to the peripherals is implemented by forwarding I/O memory accesses to the actual device.
- **PE/PM: Partial Emulation with Peripheral Modeling.** The firmware is emulated and peripheral interaction is handled by mimicking peripherals behavior with dedicated scripts inside Avatar, which allows to conduct experiments without having a physical device present.
- **FE: Full Emulation.** Both the firmware and its peripherals are fully emulated inside PANDA.

For configurations *PE/MF*, *PE/PM*, and *FE* we use a snapshot of the device's state taken *after* initialization as starting point for the emulation, as described in section VI. The execution then continues inside the emulator where we implemented the different heuristics presented in section V. To estimate the performance impact imposed by each scenario, we conducted experiments in which we selectively enable one heuristic at a time.

In all our tests, the inputs to the vulnerable software are provided on a simple textual protocol which is communicated over a serial connection. On configurations *NAT* and *PE/MF* we used the real device serial port, while in *PE/PM* and *FE* the serial port of the device is either modeled or emulated, and the input is provided to the emulator with a TCP connection and written directly in the corresponding (emulated hardware) buffer.

B. Fuzzer Setup

We built our experiments on top of `boofuzz` [40], an open source fuzzer and successor to Sulley [4], which is a popular Python-based fuzzing framework. Boofuzz does not only generate and send malformed inputs, but it also allows to define target monitoring and reset hooks. In comparison to its predecessor, it allows to fuzz over user-defined communication channels and provides ready implementations for both serial and TCP-connections, making it an ideal match for our evaluation purposes.

Obviously, inputs that triggers a fault introduces a larger overhead than those that do not. Therefore, to remove this bias and make sure we can compare the results of different experiments, we instrumented the fuzzer to forcefully generate inputs which would trigger one of the inserted memory corruption vulnerabilities with a given probability. We denote

[7] Note that we chose a Type-III device because this is the most challenging case. The intuition is that if we can detect the silent memory corruptions on a Type-III device, the heuristics are likewise suitable for Type-II and Type-I devices.

this probability as P_c and conducted experiments with $P_c = 0$, $P_c = 0.01$, $P_c = 0.05$ and $P_c = 0.1$.

Furthermore, to better simulate a realistic fuzzing session, we added a simple liveness check for monitoring purposes: After every fuzz input, the fuzzer receives the response of the device and evaluate whether it matches the expected behavior. When the received response differs from the expected one, or when the connection times out, the fuzzer reports a crash and restarts the target. The fuzzer uses `boofuzz` to power cycle the physical device (*NAT*) or instructs the emulator to restart from the snapshot (*PE/MF*, *PE/PM*, and *FE*).

Note that we use the liveness checks during all experiments, even when all heuristics are enabled as there might be crashes not detected by our heuristics. However, in our experiments, with all heuristics enabled, the liveness check never detected any corruption because the heuristics of our PANDA plugins were able to detect faulty states at a earlier stage, which in turn triggered an immediate restart of the target.

C. Results

In total, we conducted 100 fuzzing sessions lasting one hour each. We monitored the number of inputs that were processed by the target (I_{tot}), the number of times a corrupting input was sent to a target (I_C), the amount of faults detected by the liveness check (D_L), and the number of faults detected by the heuristics (D_H). Additionally, we denote the number of undetected faults as (D_U). As a result:

$$I_C = D_L + D_H + D_U \approx I_{tot} * P_C \qquad (1)$$

The interested reader can find the results of the individual fuzzing sessions in Appendix A, which are analyzed thorough the rest of this section.

False Positives and False Negatives

Intuitively, the presented heuristics are not perfect and are likely to yield false positives or negatives. Interestingly, we observed only one case of false positives in the stack object tracking when, due to over-approximation, two consecutive memory writes to set two distinct but adjacent stack variables were falsely tagged as a memory corruption.

In general, we want to stress that false positives and negative rates are highly target- and implementation-dependent and a comprehensive analysis of those are out of scope of this work. Our goal is show the limitations of fault-detection on embedded devices and the feasibility of using heuristics to overcome this problem, rather than evaluating the effectiveness of a specific implementation.

Fault Detection

Table V shows which type of corruptions could be detected by the liveness check or by the individual heuristics. As we expected, fuzzing without any fault detection mechanism is largely ineffective. The liveness check alone was only able to detect the stack-based buffer overflow and format string vulnerability because, as we already described in Section III, these bugs result in the device hanging. All the other vulnerabilities, although they were triggered correctly by the fuzzer and they resulted in a successful memory corruption, were not detected by the fuzzer.

	Format String	Stack-based buffer overflow	Heap-based buffer overflow	Double Free	Null Pointer Dereference
Liveness Check	✓	✓	✗	✗	✗
Individual Heuristics:					
a) Call Stack Tracking	✗	✓	✗	✗	✗
b) Call Frame Tracking	✗	✓	✗	✗	✗
c) Stack Object Tracking	✗	✓	✗	✗	✗
d) Segment Tracking	✓	✓	✗	✓	✓
e) Format Specifier Tracking	✓	✗	✗	✗	✗
f) Heap Object Tracking	✗	✗	✓	✓	✓
All	✓	✓	✓	✓	✓

TABLE V: Artificial vulnerabilities discovered by the different heuristics.

The impact of this is shown in Figure 1, which visualizes the amount of corrupting inputs detected by the liveness check, by the heuristics (all[8] or in isolation), or that remained undetected. A closer look at the graphs shows that the combined heuristics (rightmost bar in each group) always successfully detected all corruptions, while relying on liveness checks (leftmost bar) always left a large fraction of faults undetected. Furthermore, segment tracking, as it can both detect format string and stack based buffer overflow vulnerabilities, is superseding all detections formerly done by the liveness check. This makes sense: when the device is in a strongly corrupted state, even detectable by the liveness check, it is likely that memory accesses to unusual memory locations occured.

Performance

Figure 2 shows the number of input values the fuzzed target was able to process during one hour fuzz sessions with different values for P_C. As expected, partial emulation with memory forwarding (*PE/MF*) is slowing down fuzz testing by more than one order of magnitude. This overhead is introduced by the communication between the firmware and the device peripherals, which results into frequent invocations of the orchestration features of Avatar. The major part of this overhead is due to the low bandwidth connection between Avatar and the physical device, which relies on a standard JTAG debugger connected via USB. Surrogates [28] has shown that this issue can be solved by using dedicated hardware, which would enable partial emulation at near-realtime speed.

Looking at the individual heuristics, we can observe that their overhead is negligible in the *PE/MF* scenario, where the bottleneck of forwarding of MMIO requests fully determines

[8] Note that the "Combined Heuristics" consist of heuristic *c-f*. Heuristic *a* and *b* have been disabled as they are redundant with heuristic *c*.

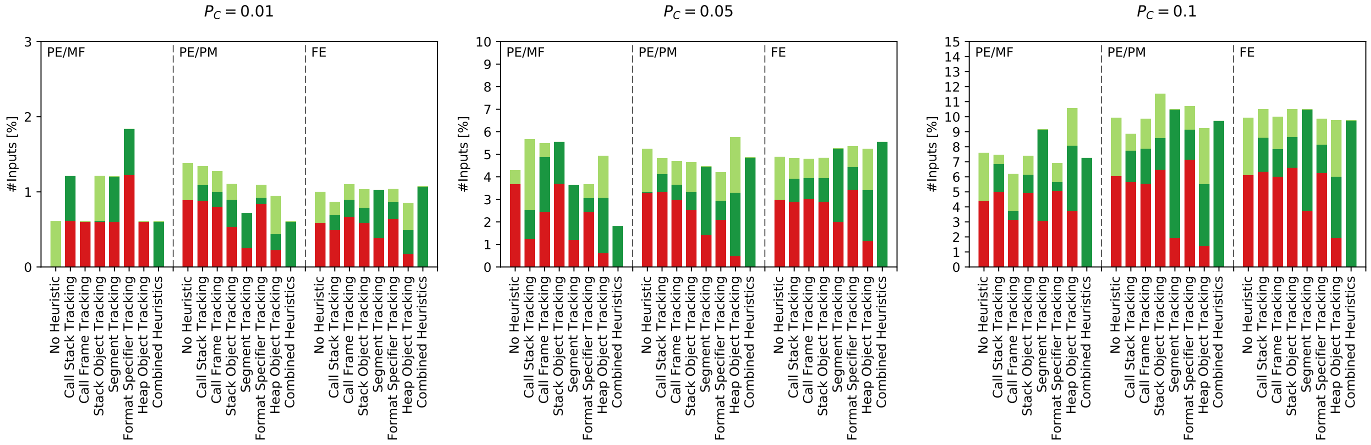

Fig. 1: Corruption detection in emulation based scenarios with distinct probabilities for the occurence of corrupting inputs P_C. 🟩 Corruptions detected by liveness checks 🟩 Corruptions detected by heuristics 🟥 Undetected corruptions

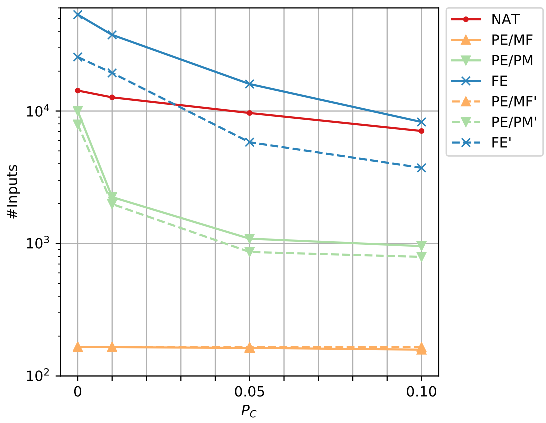

Fig. 2: Processed inputs during one hour long fuzzing sessions with no heuristics (*NAT*, *PE/MF*, *PE/PM*, *FE*) and combined heuristics (*PE/MF'*, *PE/PM'*, *FE'*) enabled.

the speed of the fuzzing experiment. However, in scenario *PE/PM* and *FE*, we can can observe a considerable slowdown (between x1.5 and x6) introduced by the heuristic analysis code for $P_C = 0$.

Another important observation is that fuzzing against a fully emulated target is significantly faster than against the physical device, as long the amount of detected corruptions is low. This is due to three main factors. First, the fact that the communication over TCP allows a higher throughput than the one over a serial port. Second, by the fact that even if the firmware is emulated, the emulator often has a much higher clock speed than (low resource) embedded devices. Third, a detected corruption is tied to a forcefully reboot of the target, which means that high P_Cs are resulting into significant time spent rebooting, rather than sending new inputs to the target.

However, the most important result of our experiments is the fact that a firmware that is executed in PANDA (full emulation) with combined heuristics enabled can be fuzzed faster than the original embedded device under realistic values for P_C. While the first can detect all classes of vulnerabilities we inserted in its code, the second needs to rely on a liveness check that can only identify two of them.

VIII. Discussion

The results of our experiments show that silent memory corruptions pose a predominant challenge for fuzzing embedded systems, as the majority of fuzzing solutions are relying on observable crashes. In particular, our tests emphasize three different aspects:

1) **Relying only on liveness tests is a poor strategy.** Fuzzing embedded systems by relying solely on liveness tests for fault detection is a poor strategy that is very likely to miss many vulnerabilities. Likewise, using only a single heuristic at a time does not guarantee the detection of more vulnerabilities. Intuitively, the highest potential for corruption detection is reached by combining several heuristics.

2) **While full emulation is the best strategy, emulators are rarely available.** Our experiments shows that, if it is possible to fully emulate the firmware of the device under test, then few selected heuristics can mitigate the lack of fault detection mechanisms. This increases the accuracy of vulnerability discovery to what we now expect when fuzzing desktop applications. While this may seem to solve the problem, full emulation is still very difficult to achieve. In particular, third party testers often lack sufficiently detailed knowledge of the hardware to implement a good emulator. Moreover, even with sufficient documentation, implementing a full emulator requires a considerable amount of manual effort.

3) **Partial emulation can lead to accurate vulnerability detection, with a significant performance impact.** When full emulation is not possible, partial emulation can lead to the same benefits in term of accuracy, at the cost of a significant slowdown of roughly one order of magnitude. In particular, partial emulation with peripheral modeling provides an interesting trade-off between vulnerability detection and fuzz speed throughput and does neither require a sound emulator nor a physical device

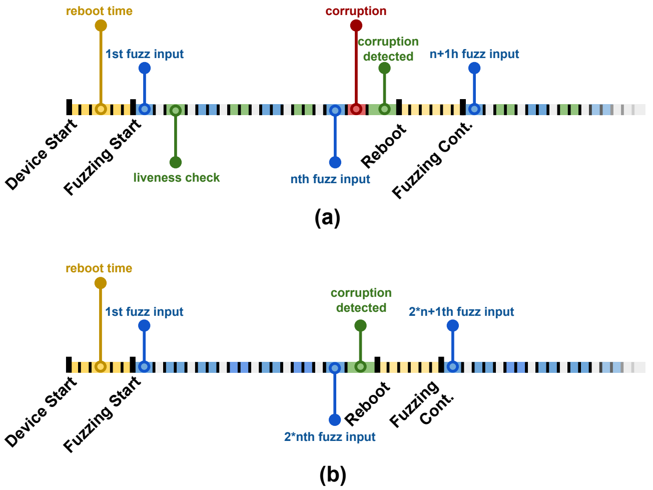

Fig. 3: Example timelines of a fuzzing session with (a) liveness checks, and (b) live detection without liveness checks.

to be present. Moreover, this setup allows to parallelize fuzzing sessions, thus making fuzz-testing embedded devices more scalable.

A further advantage of our emulation-based approach is that PANDA could also be used to record and replay the execution, which largely simplify the followup analysis to identify the root cause and possible impact when a vulnerability is detected.

Another interesting observation is that liveness checks often detect crashes due to a timeout, which significantly slows down the fuzzing experiment. In an optimal setup, where live heuristics are able to detect the majority of corruptions, the liveness check could be omitted, which could result into a significant performance improvement. This is conceptually demonstrated in Figure 3 for a simplified scenario where processing the liveness-check and processing the fuzz-input is taking the same computation time.

Finally, while our results directly impact the performance of fuzzing embedded systems, this work also applies to binary symbolic execution on embedded devices firmware (e.g., as described in [54]). An important problem of symbolic execution is the state explosion problem: with a sufficiently complex program and symbolic input the symbolic execution can rapidly reach a very large number of states which exhaust resources or takes indefinitely long time to complete. Typically, state exploration will continue until a terminating condition is found. Therefore, if the corruptions are not promptly detected, the symbolic execution could spend a significant amount of time computing useless states.

IX. Limitations

The aim of this work is to study the problem of silent memory corruptions in embedded systems and explore possible solutions. Since other aspects of fuzzing, such as the generation of inputs to discover vulnerable paths, are not specific to our domain and do not affect our study, we relied on artificial bugs to have a better control over the experiments.

Omitting real vulnerabilities from our tests may raise the question whether the insights of this work are applicable in a real world scenario. We believe this to be the case, as our work points out a fundamental (and poorly understood) limitation of the way we test embedded systems today, which is independent from the position of the vulnerabilities in the code and from the way such vulnerabilities are triggered. The behavior we observe and analyze mainly depends on the platform (OS, hardware) and on how the program was compiled (which countermeasures). The only changes when migrating from an artificial test scenario to real world software are the observed false positive and negative rates of the individual heuristics, which are, as pointed out in Section VII-C, highly implementation-specific.

Furthermore, while we initially describe several options, we then focused our study on emulation-based approaches to tackle silent memory corruptions. However, despite the increasing amount of research dedicated to this topic, the ability to fully emulate arbitrary firmware images is still an open problem. Therefore, the solution we discuss in this paper may not be applicable to all scenarios and all embedded system devices. In conclusion, our work raises awareness about the impact of silent memory corruptions on fuzzing, provides the building blocks to solve this problem, and will hopefully stimulate new research in this important direction.

X. Conclusion

In this paper, we explored the challenges in fuzzing embedded devices. We classified the types of devices and the types of memory corruption effects – which can either be *observable* or *silent*. We pinpointed that one of the predominant issues are *silent* memory corruptions. Silent memory corruptions are not specific to embedded systems. However, desktop system have a variety of protection mechanisms which are turning the majority of silent corruptions into observable crashes, while those mechanisms are often lacking on embedded devices. Hence, we studied the effects of silent corruptions on different classes of embedded devices. We evaluated a variety of approaches to mitigate this issue, and implemented a system based on Avatar and PANDA, which implements six different live analysis heuristics for partial and full emulation. We conducted experiments with this system which indicates that live analysis can improve fuzzing of embedded systems. Finally, this work shows the importance of having good emulators. Therefore, we hope that our results will stimulate more research in both improving the construction of emulators for embedded devices and in fuzzing such devices.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *12th ACM conference on Computer and communications security (CCS)*, 2005.

[2] V. Alimi, S. Vernois, and C. Rosenberger, "Analysis of embedded applications by evolutionary fuzzing," in *International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2014.

[3] M. Almgren, D. Balzarotti, J. Stijohann, and E. Zambon, "D5.3 report on automated vulnerability discovery techniques," 2014.

[4] P. Amini and A. Portnoy, "Sulley fuzzing framework," 2010.

[5] Z. Basnight, J. Butts, J. Lopez, and T. Dube, "Firmware modification attacks on programmable logic controllers," *International Journal of Critical Infrastructure Protection*, vol. 6, no. 2, 2013.

[6] A. Beckus, "QEMU with an STM32 microcontroller implementation," 2012, http://beckus.github.io/qemu_stm32/.

[7] D. L. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, 2004.

[8] D. D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[9] X. Chen, A. Slowinska, and H. Bos, "Who allocated my memory? Detecting custom memory allocators in C binaries," in *WCRE*, 2013.

[10] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," *ACM SIGPLAN Notices*, vol. 46, no. 3, 2011.

[11] Clang Users Manual, "Undefined behavior sanitizer."

[12] A. Costin, A. Zarras, and A. Francillon, "Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces," in *11th ACM symposium on Information, computer and communications security (ASIACCS)*, 2016.

[13] 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*. ACM, 2015.

[14] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-scale automated vulnerability addition," in *37th IEEE Symposium on Security and Privacy (SP)*, 2016.

[15] M. Eddington, "Peach fuzzing platform," *Peach Fuzzer*, 2011.

[16] I. D. Foster, A. Prudhomme, K. Koscher, and S. Savage, "Fast and Vulnerable: A Story of Telematic Failures," in *9th USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[17] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *31st International Conference on Software Engineering*. IEEE, 2009.

[18] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols," in *Securecomm*, 2015.

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

[20] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," in *USENIX Security Symposium*, 2013.

[21] S. Heath, *Embedded systems design*. Newnes, 2002.

[22] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan, "Recovering C++ objects from binaries using inter-procedural data-flow analysis," in *Proceedings of the ACM SIGPLAN on Program Protection and Reverse Engineering Workshop (PPREW)*, 2014.

[23] N. Kamel and J.-L. Lanet, "Analysis of HTTP protocol implementation in smart card embedded web server," *International Journal of Information and Network Security*, vol. 2, no. 5, 2013.

[24] M. Kammerstetter, D. Burian, and W. Kastner, "Embedded Security Testing with Peripheral Device Caching and Runtime Program State Approximation," in *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*, 2016.

[25] M. Kammerstetter, C. Platzer, and W. Kastner, "PROSPECT: Peripheral Proxying Supported Embedded Code Testing," in *9th ACM symposium on Information, computer and communications security (ASIACCS)*, 2014.

[26] Keen Security Lab, "Car hacking research: Remote attack tesla motors," 2016, http://keenlab.tencent.com/en/2016/09/19/Keen-Security-Lab-of-Tencent-Car-Hacking-Research-Remote-Attack-to-Tesla-Cars/.

[27] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham *et al.*, "Experimental security analysis of a modern automobile," in *30th IEEE Symposium on Security and Privacy (SP)*, 2010.

[28] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: enabling near-real-time dynamic analyses of embedded systems," in *9th USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[29] H. Lee, K. Choi, K. Chung, J. Kim, and K. Yim, "Fuzzing CAN Packets into Automobiles," in *29th International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2015.

[30] U. Lindqvist and P. G. Neumann, "The future of the internet of things," *Communications of the ACM*, vol. 60, no. 2, 2017.

[31] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN notices*, vol. 40, 2005.

[32] D. McMillen, "Security attacks on industrial control systems," IBM Security, Tech. Rep., 2015.

[33] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, 1990.

[34] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," *Black Hat USA*, vol. 2015, 2015.

[35] C. Mulliner, N. Golde, and J.-P. Seifert, "SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale," in *USENIX Security Symposium*, 2011.

[36] NCC Group, "TriforceAFL," 2017, https://github.com/nccgroup/TriforceAFL.

[37] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM SIGPLAN notices*, vol. 42, no. 6, 2007.

[38] A. Nordrum, "The internet of fewer things [news]," *IEEE Spectrum*, vol. 53, no. 10, 2016.

[39] D. Papp, Z. Ma, and L. Buttyan, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy," in *13th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2015.

[40] J. Pereyda, "boofuzz," 2016, https://github.com/jtpereyda/boofuzz.

[41] J. R., "Processor tracing," Intel Blog, September 2013, https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[42] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *Network and Distributed System Security Symposium (NDSS)*, 2017.

[43] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *USENIX Annual Technical Conference*, 2012.

[44] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: data race detection in practice," in *Proceedings of the 3rd Workshop on Binary Instrumentation and Applications (WBIA)*. ACM, 2009.

[45] K. Serebryany, "Sanitize, Fuzz, and Harden Your C++ Code," 2016.

[46] T. Shellphish, "Cyber Grand Shellphish," Phrack Papers, 2017.

[47] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A Dynamic Excavator for Reverse Engineering Data Structures," in *Network and Distributed System Security Symposium (NDSS)*, 2011.

[48] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++," in *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015.

[49] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[50] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[51] F. Van Den Broek, B. Hond, and A. C. Torres, "Security testing of GSM implementations," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2014.

[52] V. Van der Veen, L. Cavallaro, H. Bos *et al.*, "Memory errors: the past, the present, and the future," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2012.

[53] Vendicator, Stack Shield, "A stack smashing technique protection tool for Linux," 2000, http://www.angelfire.com/sk/stackshield/info.html.

[54] 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.

[55] M. Zalewski, "American fuzzy lop," 2014, http://lcamtuf.coredump.cx/afl/.

Appendix

	NAT		PE/MF		PE/PM		FE		
	D_{tot}	I_{tot}	D_{tot}	I_{tot}	D_{tot}	I_{tot}	D_{tot}	I_{tot}	
No Heuristic	–	14269	–	166	–	9997	–	53390	P_c=0
Individual Heuristics:									
a) Call Stack Tracking	–	–	–	166	–	8467	–	30418	
b) Call Frame Tracking	–	–	–	166	–	4213	–	5400	
c) Stack Object Tracking	–	–	–	166	–	8233	–	28318	
d) Segment Tracking	–	–	–	166	–	9390	–	41101	
e) Format Specifier Tracking	–	–	–	166	–	9257	–	38158	
f) Heap Object Tracking	–	–	–	166	–	9321	–	39654	
Heuristics c,d,e,f	–	–	–	166	–	7921	–	25557	
No Heuristic	43%	12681	100%	165	35%	2241	40%	37569	P_c=0.01
Individual Heuristics:									
a) Call Stack Tracking	–	–	50%	165	34%	2386	42%	25059	
b) Call Frame Tracking	–	–	0%	166	37%	2512	38%	18289	
c) Stack Object Tracking	–	–	50%	165	52%	1894	42%	23213	
d) Segment Tracking	–	–	50%	166	64%	2372	61%	31768	
e) Format Specifier Tracking	–	–	33%	163	23%	2382	38%	31123	
f) Heap Object Tracking	–	–	0%	166	76%	1794	79%	28564	
Heuristics c,d,e,f	–	–	100%	166	100%	1985	100%	19387	
No Heuristic	39%	9663	14%	163	36%	1087	38%	15985	P_c=0.05
Individual Heuristics:									
a) Call Stack Tracking	–	–	77%	159	30%	1143	39%	15604	
b) Call Frame Tracking	–	–	55%	164	36%	874	37%	15212	
c) Stack Object Tracking	–	–	33%	162	45%	905	40%	15537	
d) Segment Tracking	–	–	66%	165	68%	987	62%	9688	
e) Format Specifier Tracking	–	–	33%	164	50%	1192	35%	15909	
f) Heap Object Tracking	–	–	87%	162	91%	852	78%	7772	
Heuristics c,d,e,f	–	–	100%	165	100%	863	100%	5813	
No Heuristic	40%	7067	41%	158	38%	956	38%	8267	P_c=0.1
Individual Heuristics:									
a) Call Stack Tracking	–	–	33%	161	36%	813	39%	7773	
b) Call Frame Tracking	–	–	50%	161	43%	811	39%	7773	
c) Stack Object Tracking	–	–	33%	162	43%	850	36%	8122	
d) Segment Tracking	–	–	66%	164	81%	868	64%	5048	
e) Format Specifier Tracking	–	–	27%	159	32%	908	36%	8612	
f) Heap Object Tracking	–	–	64%	161	84%	779	80%	4387	
Heuristics c,d,e,f	–	–	100%	165	100%	793	100%	3727	

Appendix A: Number of inputs sent to the target during a fuzzing session I_{tot} and vulnerability detection efficacy D_{tot} for different corruption probabilities P_C in the four target setups *Native (NAT)*, *Partial Emulation with Memory Forwarding (PE/MF)*, *Partial Emulation with Peripheral Modeling (PE/PM)* and *Full Emulation (FE)*.