Fuzzing the Internet of Things: A Review on the Techniques and Challenges for Efficient Vulnerability Discovery in Embedded Systems

Maialen Eceiza, Jose Luis Flores, and Mikel Iturbe

***Abstract*—With a growing number of embedded devices that create, transform and send data autonomously at its core, the Internet-of-Things (IoT) is a reality in different sectors such as manufacturing, healthcare or transportation. With this expansion, the IoT is becoming more present in critical environments, where security is paramount. Infamous attacks such as Mirai have shown the insecurity of the devices that power the IoT, as well as the potential of such large-scale attacks. Therefore, it is important to secure these embedded systems that form the backbone of the IoT. However, the particular nature of these devices and their resource constraints mean that the most cost-effective manner of securing these devices is to secure them before they are deployed, by minimizing the number of vulnerabilities they ship. To this end, fuzzing has proved itself as a valuable technique for automated vulnerability finding, where specially crafted inputs are fed to programs in order to trigger vulnerabilities and crash the system. In this survey, we link the world of embedded IoT devices and fuzzing. For this end, we list the particularities of the embedded world as far as security is concerned, we perform a literature review on fuzzing techniques and proposals, studying their applicability to embedded IoT devices and, finally, we present future research directions by pointing out the gaps identified in the review.**

***Index Terms*—Embedded system, fuzzing, Internet-of-Things (IoT), software testing, vulnerabilities**

I. Introduction

THE Internet-of-Things (IoT) is the novel networking paradigm where heterogeneous computing devices, known as IoT devices, interact between them with little to no human intervention to collaborate towards a common goal [1]. Thanks to this total inter connectivity, IoT devices can continuously create and stream information that operators can leverage and provide value-added services on top of it in areas such as industry, smart cities/homes, security applications, health care, etc., as it is shown in Fig. 1. Examples of such services include predictive maintenance, precision healthcare, security monitoring, smart crop management or advanced control of a production process.

Since its recent inception, the IoT has undergone a near exponential growth and by 2025, the world will have 75 billion IoT devices [2], if the actual grade of growth continues. In other words, in five years time the number of IoT devices will be doubled, effectively duplicating the size of the current IoT.

IoT devices, while heterogeneous in nature, are rather resource-constrained when compared to general-purpose computing devices, such as laptops or workstations. The reason for this constraint is that IoT devices or *things* are embedded systems based on a micro controller that can transmit and receive information [3], mainly interacting with the physical world using peripherals and optimizing resource usage. While this constraint allows to build cheaper IoT devices, easing IoT adoption, it also means that devices often lack functionalities that are considered non essential, such as security. Moreover, this constraint also translates in the difficulty or impossibility of conducting frequent upgrades on the system securely [4].

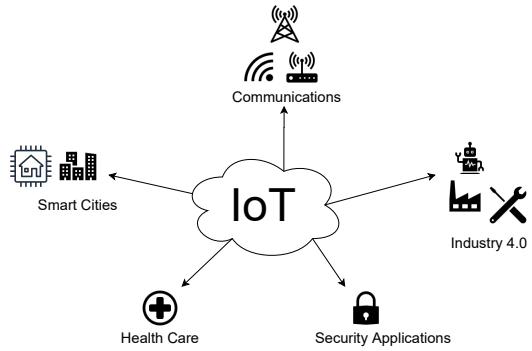

Fig. 1. Internet-of-Things systems application domain

As a consequence, IoT device security is a unique challenge, vastly different from the traditional security paradigm in the Information Technology (IT) domain, as the devices and systems differ in nature and capabilities. This challenge has yet to be properly addressed, as the increasing number of attacks that target IoT devices shows: in the first half of 2019 alone, more than 100 million smart devices were attacked [5]. These attacks range are varied in nature, ranging from physical attacks to crypto-analytical ones [6] and their effects can be both far-reaching, as shown by the infamous Mirai Distributed Denial-of-Service (DDoS) attack in 2016 [7], as well as potentially deadly, as they can target IoT devices inside critical systems such as cardiac pacemakers [8] or cars [9]. These attacks often exploit vulnerabilities, that is, security flaws in the software that allow attackers to gain control of the system through their exploitation. In their survey related to IoT

M. Eceiza and J.L. Flores are with the Department of Industrial Cybersecurity, IKERLAN Technology Research Center, Basque Research and Technology Alliance (BRTA), P. J. M. Arizmendiarrieta 2, E-20500 Arrasate-Mondragón, Spain (e-mail: meceiza@ikerlan.es;jlflores@ikerlan.es).

M. Iturbe is with the Department of Computing and Electronics, Faculty of Engineering, Mondragon Unibertsitatea, Goiru 2, E-20500 Arrasate-Mondragón, Spain (e-mail: miturbe@mondragon.edu).

vulnerabilities, Neshenko et al. [10] also identified the unmet challenge of vetting deployed IoT code, posing an important risk for IoT security as a whole. It is, thus, critical to reduce the number of vulnerabilities to a minimum in order to hinder exploitation.

The aforementioned particularities of embedded IoT devices (lack of resources and difficult to update) suggest that in order to be secured, the most cost-effective approach is to minimize their vulnerabilities before their deployment to the field. Therefore, it is a matter of discovering and patching as many security vulnerabilities as possible while the system is still under development. While many approaches for vulnerability discovery exist, fuzzing remains as one of the main techniques for this endeavour. According to Manes et al. [11] fuzzing remains popular due to its simplicity, its low barrier to deployment, and its vast amount of empirical evidence in discovering real-world software vulnerabilities. In essence, fuzzing consists of repeatedly feeding deliberately malformed inputs to a target (a device or a program) known as System Under Test (SUT) in order to provoke crashes and finding the vulnerabilities that cause them. It has also been defined as one of the effective ways to identify software vulnerabilities by testing [12]. Fuzzing has developed wide applications since its inception in 1990, when Miller et al. [13] developed the first fuzzing tool [14]. Since then, numerous proposals have been developed combining different techniques for fuzzing (refer to surveys [15], [16], [17], [11] for a wider analysis on the topic), improving the general performance both in number of found vulnerabilities as well as the time to do so.

Therefore, it seems natural to link fuzzing and embedded IoT, by integrating fuzzing as part of the embedded software development life cycle to hunt as many vulnerabilities as possible before the system is finally released. The suitability of this match was also pointed out by Muench et al. [18] and also by the ISO/IEC 62443-4-1 standard, that states that fuzzing is a necessary step for embedded product certification [19], [20], [21]. Even if fuzzing has been used effectively on IT systems to test traditional software and remains widely used, its adoption has not been that widespread inside the the IoT or embedded environments [22], [23], [24]. The main reason for this lack of adoption resides in the differences of embedded systems when compared to their IT counterparts, particularly the scarcity of resources available and the lack of informative system responses. These differences render using general-purpose fuzzers against embedded devices difficult, as they risk overflowing the system with a higher number of inputs that the systems can handle. Moreover, fuzzers catered to embedded systems need to be able to restart the SUT in order to re-establish a clean state for the next test case [18], which is not as simple as restarting a process as it happens in IT systems. Early examples of real-life embedded fuzzing include the discovery of vulnerabilities in smartphones by fuzzing SMS messages [25], testing GSM implementations [26], fuzzing the CAN protocol in safety-critical applications such as in vehicles to unlock doors [27], or in payment systems by fuzzing credit cards to exceed the limits of payments [28]. Fuzzing is a promising technique for embedded systems, as it allows to find vulnerabilities without knowing its internal operation and only focusing the I/O content of the device. However, as mentioned earlier, using fuzzing to test embedded systems and IoT devices, while promising, also presents its own set of challenges.

This survey paper aims to provide the necessary foundation to enable IoT device fuzzing research, by reviewing, analyzing and discussing the existing literature on fuzzing approaches from the point of view of the embedded and IoT world, as well as identifying future research areas. As such, this survey complements past fuzzing surveys ([15], [16], [17], [11]) by basing its analysis in an application field not considered previously. In particular, the paper presents the following contributions:

- A review of the literature on different fuzzing proposals
- A comparison and discussion of said proposals based on their applicability to embedded systems.
- The identification of different future research lines related to fuzzing and embedded systems.

The rest of the paper is organized as follows: Section II introduces embedded systems and their particularities and challenges regarding security. Section III presents fuzzing and the different existing approaches when comparing them. Section IV details the current necessities and challenges of fuzzing embedded systems. Section V compares existing fuzzing approaches and outlines the features an embedded system fuzzer should have. Section VI draws some future research lines evolved from the necessities identified in the previous analysis. Finally, Section VII concludes the paper.

II. Embedded Systems

While embedded or Cyber Physical Systems (CPS) are not equivalent terms for the IoT [29], it is true that IoT devices can be considered embedded systems [3], [30]. Furthermore, according to the IEEE document titled *Towards a definition of the Internet of Things (IoT)* [31], the main difference between a CPS and an IoT device is that the the CPS does not have the requirement of being connected to Internet. This is, precisely, a key factor in terms of security when a *thing* is connected to Internet, as it vastly expands its attack surface.

According to the same document [31], in order for a system or *thing* to be considered an IoT device, it must contain the following set of features (depicted in Figure 2):

- Interconnection of things
- Programmability
- Self configurability
- Connection of Things to the Internet
- Embedded Intelligence
- Uniquely Identifiable Thing
- Interoperable Communication Capability
- Sensing/Actuation Capability
- Ubiquity: Anywhere, Anytime

Depending on the complexity of the IoT scenario at hand, the capabilities and nature of IoT devices also also range from full-fledged computer hosts to small and very elemental embedded devices. In this sense, embedded devices are located in the opposite end from computer hosts which are able to perform very complex tasks at the same time.

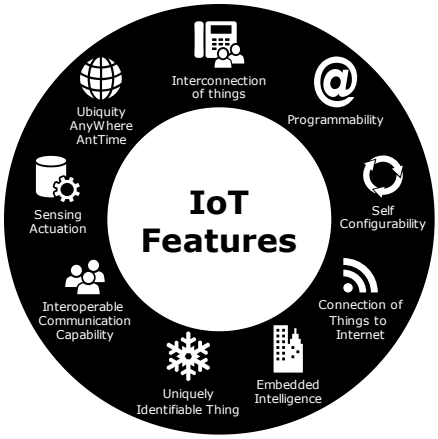

Fig. 2. Features that a system must contain to be considered an IoT device

In essence, embedded systems are dedicated software and hardware solutions that have been designed to perform a specific function, generally interacting with the physical world using their peripherals [32]. These systems are generally part of a larger structure, such as the IoT, where they can be used in a wide range of applications. It is, precisely, the varied and specific nature of their applications what yields the complexity of these systems when trying to classify them. On one end, for the more simple applications, devices with low memory and processing power are used, such as STM32 micro controllers [33]. On the other end, for the more complex tasks, the embedded systems that are used are not that constrained and are nearer from IT standards, even to the point to be able to substitute a computer in its most simple tasks, such as in the case of the popular Raspberry Pi 4 [34]. It is necessary to note that the hardware and software of the embedded device is designed according to the needs and it is not oversized, since the size, consumption and cost should be minimal to perform the task at hand [35]. For simpler tasks, the more constrained devices operate without an operating system (OS), while for more complex tasks, embedded systems ship OSs similar to the ones that can be found on desktop PCs [18]. Apart from the computing power at hand, time is also a critical factor when considering embedded devices. Most devices need a deterministic response to real-time events [36]. These systems are named *real-time* embedded devices. Next, we present a classification of embedded systems according to their capabilities and specific security concerns related to this field.

A. Classification of Embedded Systems

The aforementioned diversity of embedded systems makes it difficult to designed clear classification, as different criteria can be used to establish this. Some classifications are based on the functional requirements of the system (real-time, stand alone, networked, or mobile), the complexity of the micro controller (small, medium, or sophisticated) [37], the application filed (manufacturing, security, transport, etc.) or the type of the operating system (OS) the system is running. In this paper we will focus on the classification according to the OS proposed by Muench et al. [18], relevant to the field of fuzzing:

- **Systems running general-purpose OSs.** Embedded systems with general-purpose operating system are used to manage operator interfaces, databases and general-purpose computing tasks [38]. These OSs are used in some high-end embedded systems, but they need to be customized to the corresponding hardware and maintaining the main features/services of the OS. Those systems are multi-core and they have more than a gigabyte of RAM memory. An example of such OS is Busybox, a OS which is based in UNIX, providing some of its utilities in a single small executable [39].
- **Systems running embedded OSs.** Embedded OSs are designed to be customized in specific hardware losing some of the features/services. As a counterpart, the embedded OS is more efficient and reliable. The hardware features of the devices that use this kind of OS are the following ones: between a megabyte and a gigabyte of RAM memory and with one or two core. In addition, the embedded OS is configurable for the needs of the device and it is very useful when the device has limited processing power [40]. Example of this kind of operating systems are FreeRTOS [41] or VxWorks [42], which are real time operating systems.
- **Systems with no OS.** The limitations of the hardware such as a memory limited to a few megabytes or the presence of single core makes impossible to use a full-fledged OS. However, the use of a simple scheduler or a debug monitor and the corresponding application is enough to carry out a simple tasks. Usually these devices have a single loop control and the peripherals trigger interruptions [18]. Some examples of embedded devices without operating system are WiFi cards or GPS dongles, and sometimes the code is based on OS libraries such as TinyOS [43], which provide OS-like functionalities without having to ship an OS.

B. Security on Embedded Systems

Nowadays, information security is one of the main concerns of general-purpose systems where many resources are being invested in different technologies to protect the confidentiality, integrity and availability of these systems at different layers [44]. It is easy to find technologies to protect the perimeter, to inspect internal traffic, and also to detect malware in the operating systems among others aspects. More recently, the concept of zero-trust [45] is becoming more prevailing, that is, the concept that every system should be able to protect themselves without the help or assistance of external security systems. The beneficiaries of these technologies are the general-purpose systems which still have high enough resources to integrate these technologies without any operation impact, including the capability to patch and/or update systems. However, the landscape in embedded systems is quite far from this situation, these systems have been traditionally designed with functional requirements in mind and the security has not been considered in the design process, only aspects such as cost, performance or power have been traditionally taken into account. Furthermore, the incorporation of security

technologies is very limited or even impossible due to the hardware constraints of the embedded systems. As a consequence, the IoT suffers from a wide range of security vulnerabilities, exhaustively described by Neshenko et al. [10] in their survey.

In order to increase IoT security, it is necessary to secure the weakest link, that is, the embedded systems who lack the proper mechanisms to mitigate security threats. For this end, designers should consider not only new system architecture designs with security in mind, but also complementing them by adding security features such as hardware security modules, cryptographic algorithms, security protocols, secure configurations and the latest secure and stable versions of software libraries, kernels, etc. The inclusion of these new features also requires the inclusion of the corresponding maintenance tasks as well as the management ones during the lifetime of the device. This represents a challenge in terms of the market. Whatever advance in the development of secure products represents an advantage in commercial terms.

In this sense, one of the most consuming tasks in the development life-cycle is the testing process due to the difficulty of finding misconfiguration, bugs or vulnerabilities. Therefore, it is natural to find "low-hanging-fruit" vulnerabilities, i.e. vulnerabilities that are easy to find and exploit due to insufficient testing. As a consequence, traditional testing methodologies (functional tests and safety-related tests) are being complemented with new testing approaches that are being able to find more vulnerabilities in shorter time, and fuzzing is one of the most suitable technologies for this task [18]. Indeed, according with the standard IEC 62443-4, in particular "IEC 62443-4 Practice 5, SVV-3 Vulnerability testing", the application of fuzzers should be carried out in external interfaces without additional information the section of the standard corresponding to the component development considers the fuzzing as mandatory step in every new development.

However, when facing a constrained environment such a embedded system, the only viable manner to find vulnerabilities remains external, out-of-device testing, as it is not possible to run extra security resources on top of the device. Therefore, fuzzing presents itself as an interesting alternative, where it has the ability of testing an embedded SUT without needing to run extra software on the device nor knowing its internal workings. However, the specific nature of embedded systems makes fuzzing more complex, less efficient and prone to errors than in classical general-purpose systems. There are three main resources general-purposes have and embedded systems generally lack that impact fuzzing performance directly.

Firstly, classical fuzzing systems rely on the ability to detect crashes. In general-purpose IT systems, the OS provides joint security mechanisms that prevent a program to perform an invalid operation (e.g. trying to access a memory address not belonging to the same process), resulting in a crash. Some of these mechanisms are:

- **Address Space Randomization (ASLR)**. This mechanism arranges at random the address space positions of data areas, including the base of the executable and the positions of the stack, heap and libraries. In this way, the classical computation of memory addresses by malware pieces does not produce the desired effect, that is, to exploit a vulnerability in very precise way.
- **Code Integrity Guard**. It controls the arbitrary code generation by reinforcement signature constraints for loading binaries. This technique is primarily supported by cryptographic mechanisms, particularly, digital signatures.

TABLE I
HARDWARE PROTECTION MECHANISMS BY HARDWARE FAMILIES [46]

Hardware Family	MPU	MMU	DEP	CFI
ARM 1 to ARM 7	✖	✖	✖	✖
ARM Cortex R	✔	✖	✖	✖
ARM Cortex M	∼ [1]	✖	✔	✖
PIC 10 to PIC 24	✖	✖	✖	✖
Intel MCS-51	✖	✖	✖	✖
Infineon XC88X-I	✖	✖	✖	✖
Infineon XC88X-A	✖	✖	✖	✖

[1] Supported by some micro controllers of the family

Secondly, the hardware that is present in general-purpose systems also provides security mechanisms that jointly with the OS prevent programs to perform invalid operations (e.g. trying to execute memory pages that should contain data). Some of these mechanisms are:

- **Control Flow Integrity (CFI)**. This technique instruments the code by adding lightweight security code with the aim of controlling the validity of the origin of the call. An implementation example is the Control Flow Guard present from Windows 8.1.
- **Data Execution Prevention (DEP)**. This is a system-level protection feature that enables the system to mark pages of memory as non-executable. This mechanism prevent the execution of code from the marked memory pages making harder the exploitation of buffer overflows.
- **Memory Management Unit (MMU)**. It performs the translation of virtual memory addresses to physical addresses that jointly with the operating system can prevent the access to specific pages corresponding to other privileged processes.
- **Memory Protection Unit (MPU)**. This mechanism prevents a process from accessing memory that has not been allocated to it. The MPU also allows to define access permissions and attributes to specific regions of memory, monitoring any kind of access to these and triggering an access violation exception when is detected.

Finally, general-purpose systems' hardware has enough resources to support maintained load without being affected, which means that the fuzzer can operate at full performance without affecting primary functions of the device.

All these characteristics facilitate the use of fuzzers in general-purpose IT systems. However, these mechanisms are not commonly found in embedded systems as a consequence of their constrained environment. As an example, Table I shows the availability (or lack thereof) of hardware-based protection mechanisms in different micro-controller families.

As a consequence, when a program running on an embedded device crashes, the behaviour of the system as a whole can vary abruptly, ranging from an increase in response time to a full crash of the whole embedded system, leaving it completely

unresponsive. Such uncertainty hinders the ability of a external fuzzer to infer a particular type of error.

III. Fuzzing

Fuzzing or, fuzz-testing, is the method that finds vulnerabilities and bugs by inserting specially crafted inputs into a target, named System Under Test (SUT) [47], [48]. These specially crafted inputs trigger non-expected behavior in the SUT, and allow to find bugs such as faulty memory violations, assertion violations, incorrect null handling, deadlocks, infinite loops, undefined behaviors or incorrect managements of other resources. When compared to other vulnerability finding strategies such as code inspection or reverse engineering, fuzzing has the advantage that can be performed at large-scale and unattended, as the fuzzing process is usually automated. This process is classically divided in the following phases:

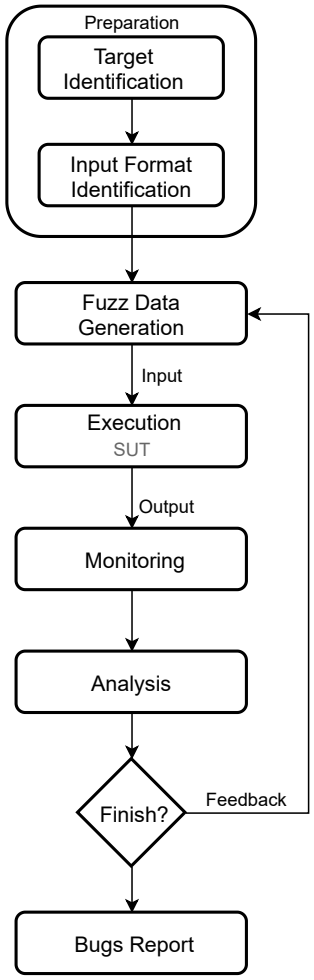

Fig. 3. Phases of the fuzzing process

Preparation. This is the first phase and it is focused on the identification and specification of the format of the inputs and the outputs of the SUT. Based on this specification is possible to reduce the possibility of generating initial invalid fuzz data and create valid and precise inputs.

Fuzz data generation. In this phase, the input data of the SUT are generated, taking into account the input format identified in the previous phase. Usually, the generation of the data is done by altering specific data. Firstly, a seed (i.e. original input) is selected; next, the fuzzer creates new fuzz data by inserting small modifications into the seed. This generation is a critical aspect in fuzzing, as the performance on vulnerability search is directly related to the quality of these inputs [11], [15], and particularly, the choice of the initial seed [49].

Execution. In this phase the previously generated data are sent to the the SUT, by using the specified media (communication packet, input file, environment variable, analog input etc.).

Monitoring. In this phase the outputs of the SUT as well as the behaviour are monitored in order to detect unexpected outputs or crashes that could be related to triggering a vulnerability. This is one of the core basis of fuzzing, an absence of a monitoring functions may result in poor results in terms of vulnerabilities findings.

Analysis In this phase, the crashes and the provided inputs from the monitoring phase are analyzed with the aim of determining which test cases cause vulnerability or abnormal behaviour triggering.

Bug reporting. Finally, when the fuzzing process is finished, the vulnerabilities that are found in the monitoring phase and are later analysed are reported. In addition, the triggering test cases are saved in order to repeat the test and reproduce the crash, allowing to analyze the vulnerability more deeply.

Figure 3 shows the process of fuzzing. The process consists of six different stages and, in order to perform them the fuzzers needs to formed by at least three elements or subsystems:

Fuzz Data Generator Based on the knowledge of the SUT, this subsystem generates the fuzz data that will be sent to the SUT. The effectiveness and accuracy of this subsystem will depend on two main factors, the prior information of the SUT and the feedback received from the SUT after conducting a test.

Protocol interface It collects the data that the generator has created and to be sent to the SUT to be executed.

Monitor It receives the outputs of the SUT with the purpose of finding unexpected outputs that could relate to vulnerabilities.

Although these three elements are the bare minimal components that all fuzzers have, more complicated setups can also add more elements depending the task on hand, such as SUT instrumentation or result analyzer.

A. Fuzzing Taxonomy

The classification of fuzzers can be performed according to their behavior in each of the fuzzing phases: prior knowledge of the target and their input (*Preparation*), data generation approaches (*Fuzz data generation*), data generation intelligence based on testing feedback (*Analysis* and *Fuzz data generation*), the exploration strategy (*Analysis* and *Monitoring*) and the used technique (*Execution*). In the following sections, we outline several criteria used for fuzzer classification. However, it is important to note that a single criterion is not enough to

identify and characterize a fuzzer, but rather a combination of all of them (e.g. a black-box, mutation-based fuzzer).

1) Black-box, White-box or Grey-Box fuzzing: This criteria considers the prior knowledge available regarding the SUT, based on this the fuzzing techniques can be divided into three main categories:

- **Black-box** testing refers to the case where the fuzzer has no access to the source code nor the internal logic of the SUT [24]. Most frequently, black-box fuzzers use random mutation limited by some rules, aimed to create valid inputs that the SUT will accept [50]. If those inputs were to be generated in a completely random manner, the SUT would discard the majority of the generated inputs and the performance of the test would decrease, as most of the inputs would fail to even run [16]. The main advantage of black-box testing is that no source code is needed, therefore, it can be applied against virtually any target. However, it is difficult to determine the code coverage and they are not use to find errors caused by complex attacks. Initially, all fuzzers were black-box, because to use those fuzzers was not necessary to know the source code and the internal working of the SUT [11]. One representative black-box fuzzing tool is zzuf [51].
- **White-box** testing makes use all of the information about the internal logic, that is, the source code [15][24]. The main advantage of white-box testing is that it can cover the code completely, as the source code and the internal logic is known to the fuzzer [16], it can infer which part of the code has been executed. In addition, before started the testing it is necessary to analyze all the information, and techniques such as instrumentation are usual, for this reason is more difficult than black-box testing to start with the testing. Nevertheless, white-box fuzzers require code access, which is not available in many cases (e.g. when auditing third-party software) and are also known to cause false positives [14]. Guided coverage and the dynamic symbolic analysis [50] are the most used methods in white-box fuzzing. SAGE [52] and Dowser [53] are two popular fuzzing white-box tools.
- **Grey-box** testing has partial access to the internal logic of the SUT. Its execution logic is similar is similar to black-box fuzzers, but it can leverage some limited information (often gathered through instrumentation) about the SUT to improve fuzzing performance and coverage [16], [11]. Nowadays, grey-box testing is the most widespread method, as it does not require full source-code access but it is able to infer information that makes it more efficient than black-box testing [14], [47]. The most well known grey-box fuzzer is the American Fuzzy Lop (AFL) [54].

2) Input generation: As stated previously, fuzz data generation has a central role in the performance of fuzzing, as it is directly related to the quality of the test. According to this, fuzzing approaches can be grouped in two main categories:

- **Mutation-based** approaches generate the new inputs from the previously generated test cases. The first time it is necessary to provide a seed to generate the next test cases, that are created by tweaking the original seed. Each tweak is named as *mutation*. The selection of the seed will condition the quality of the test cases and, thus, the code coverage. A better seed will yield a wider coverage. For this generation strategy, it is not necessary to know the specifications of the input data or protocol, as mutations alter the original seed without checking whether it complies with a specific syntax. Therefore, this method is particularly useful when input specifications are complex and data collection is accessible, as it is possible to form a seed from a wide range of recorded inputs specifying its format. When using mutation, there are two key decisions that can alter its performance: how to perform the mutation and which of the newly created values are used for the next mutation. In the first case, the mechanism to perform the mutation decreases the efficiency of the fuzzer when is blind, that is, no feedback has been considered and it is not possible to know whether the mutation strategy is the correct one [15]. The second factor is strongly related with the first one, because the absence of feedback information does not provide any clues to select a candidate fuzz data or other. Moreover, mutation is a never-ending process, requiring the fuzzer to also specify when to stop the test, which is not a simple task. There are different strategies for input mutation, and improving mutation performance is an active research field [55], [56], [15], [11]. One of the main performance problems of this method is that the seed size increases continuously, since the previous samples are joined to create the new test cases; bigger seeds cause the fuzzer to slow down. Some well-known mutation-based fuzzers are AFL [54], Angora [57] and VUzzer [47].
- **Generation-based** method uses the a set of specifications on the SUT inputs to generate new fuzz test data. In contrast to mutation-based generation, it is necessary to know the syntax of the SUT inputs, including their format and used protocol. Generation-based fuzzers are used when it is vital to provide valid input for a successful test (i.e. when the input format is constrained and random mutation would render most of the created test cases useless). On the one hand, generation-based fuzzers are faster than their mutation-based counterparts, because their seed size does not increase over multiple executions. On the other hand, modelling complex or unknown input protocols in a set of specifications might prove a challenging endeavour, and mutation-based counterparts are better suited for this task. Sulley [58] and Peach [59] are two of the most well-known generation-based fuzzers.

3) Fuzzer intelligence: The fuzzer's intelligence is related to its ability to generate new input data taking into account the feedback it receives after an execution of a test. The feedback helps to improve the new input generation, because can be used to decide which part of the test case should be modified, and how to modify it. In other words, it serves to generate more inputs that trigger new execution paths in the SUT [60]. According to their implemented intelligence, fuzzers can be defined either as *smart* or *dumb* [15].

- **Smart** fuzzers adapt the generation of test cases depending on the information they receive from the output and behaviour of the SUT, by first learning the behavior of the SUT (what is the output, has there been a crash etc.) how the test case has affected this behavior and then deciding how to generate the new test cases [61]. The smart fuzzers are more effective in detecting vulnerabilities [15], [62] as they require to execute fewer test cases. Learn&Fuzz [63], IoTFuzzer [64] and Peach [59] are examples of smart fuzzing.
- **Dumb** fuzzers do not consider feedback from previous executions as inputs for new data generation. As they do not have to receive feedback, analyze it and act based on it, dumb fuzzers are faster test executors than their smart counterparts, but less effective when considering vulnerability search [15]. A popular example of dumb fuzzing is zzuf [65].

4) Exploration strategy: Code exploration strategy refers to the method fuzzers use to maximize the covering of different parts of the code. The code coverage of the fuzzer is dependent on this strategy [15], so it is an important criterion for performance. It is important to note, that only white and grey box fuzzers can be classified according to their exploration strategy, as having information about the SUT and program, even partially is needed to be able to define a strategy. Therefore, white-box and grey-box fuzzers are divided in two main categories:

- **Covered-based** fuzzers maximize code coverage with the support of analysis techniques. The fuzzers with high coverage find more bugs and they try to maximize as many execution paths as possible with the minimum number of inputs [47], [12], [66]. Therefore, covered-based fuzzing is an efficient and effective method, as fewer test cases have to be executed [15]. One of the well-known covered-based fuzzers is SAGE [67], [52].
- **Directed** fuzzers aim to audit specific parts of the code and paths of the SUT. With this type of fuzzers, it is possible to direct the fuzz test to relevant parts of the SUT (e.g. code changed in a update, or critical parts of the application) allowing to gather faster results [68], [66]. Directed fuzzers allow to redirect test execution to avoid repeating paths and cover all the code, if necessary [15]. Most directed fuzzers are usually white-box fuzzers based on symbolic execution and oriented to generate test inputs [69]. Dowser is an example of directed fuzzer [70].

B. Fuzzing Techniques

As we have described in the previous section, the processing structure of a fuzzer involves many stages. Since the conception of the fuzzer different approaches have been focused in increasing the performance of the fuzzer by improving specific phases in the search of vulnerabilities. This motivated a new line of research in security for evolving every aspect of the fuzzer that began in 1990.

This history of the fuzzer evolution is consistent with the evolution of the number of Common Vulnerabilities and Exposures (CVEs) per year as shown in Fig. 4 and the interest in this type of approaches. Indeed, the evolution of fuzzers is associated with the capability of automate the search of vulnerabilities which shows that at the beginning the number of CVEs was relatively low and the presence of fuzzers was sporadic. However, the great of evolution of fuzzers seems to respond to the increase of vulnerabilities published from 2005 where represented a quantitative leap compared to previous years reaching more than 4000 vulnerabilities per year in the best case and more than 15000 vulnerabilities per year [71]. The evolution of the fuzzers shows that these have been integrating more complex and modern approaches such as those based on machine learning and genetic algorithms.

In this section we will review from the first technique proposed in 1990 to the most modern approaches that have been incorporated in different stages to improve the results.

- **Random mutation**. This was the first technique used in fuzzing that is categorized as a black-box technique to generate fuzz data without intelligence and strategy (see fuzz [13] and zzuf [65]). The operation of this technique requires an initial seed, and a valid input for the SUT. Based on this seed and the initial valid input, the mechanism for generating new fuzz data works by selecting at random specific field data, and then modify this field data at random, this process is repeated in each round of the fuzzer. The main advantages of this technique are the simplicity for generating fuzz data, it is not necessary to understand the data structure, and the performance obtaining initial promising results. However, the absence of a intelligence has a main drawback: the low performance. This technique has a low capability to reach complex paths which limits the power of finding new vulnerabilities [15], [24]. This technique can be considered as baseline for assessing new fuzzing techniques in terms of performance.
- **Grammar representation**. This technique is a black-box technique to generate fuzz data without intelligence and strategy but with a significant difference, the generation of data follows a set of grammar rules for decreasing the rate of invalid data at the beginning [56]. This technique requires a seed, but instead of having an initial valid input, the (initial) data are generated according to the grammar rules that check that the basic data structures are fulfilled and therefore the are no problems for beginning the process of fuzzing. Once the process has started, in each generation of data, the field data are selected at random but fulfilling the grammar rules, then the candidate fuzz data are generated at random. The result is an increase of performance compared to a completely blind technique such as the previous one because this can deal better with complex structures [63]. However, this technique has also several drawbacks such as the need to create the grammar rules by hand which requires time and knowledge [16], [72] and besides this it is subject human errors. On contrary, although the use of grammar rules is beneficial during the process has an negative impact in performance for generating new fuzz data, and the rate of fuzz data per second is lower than the previous technique limiting

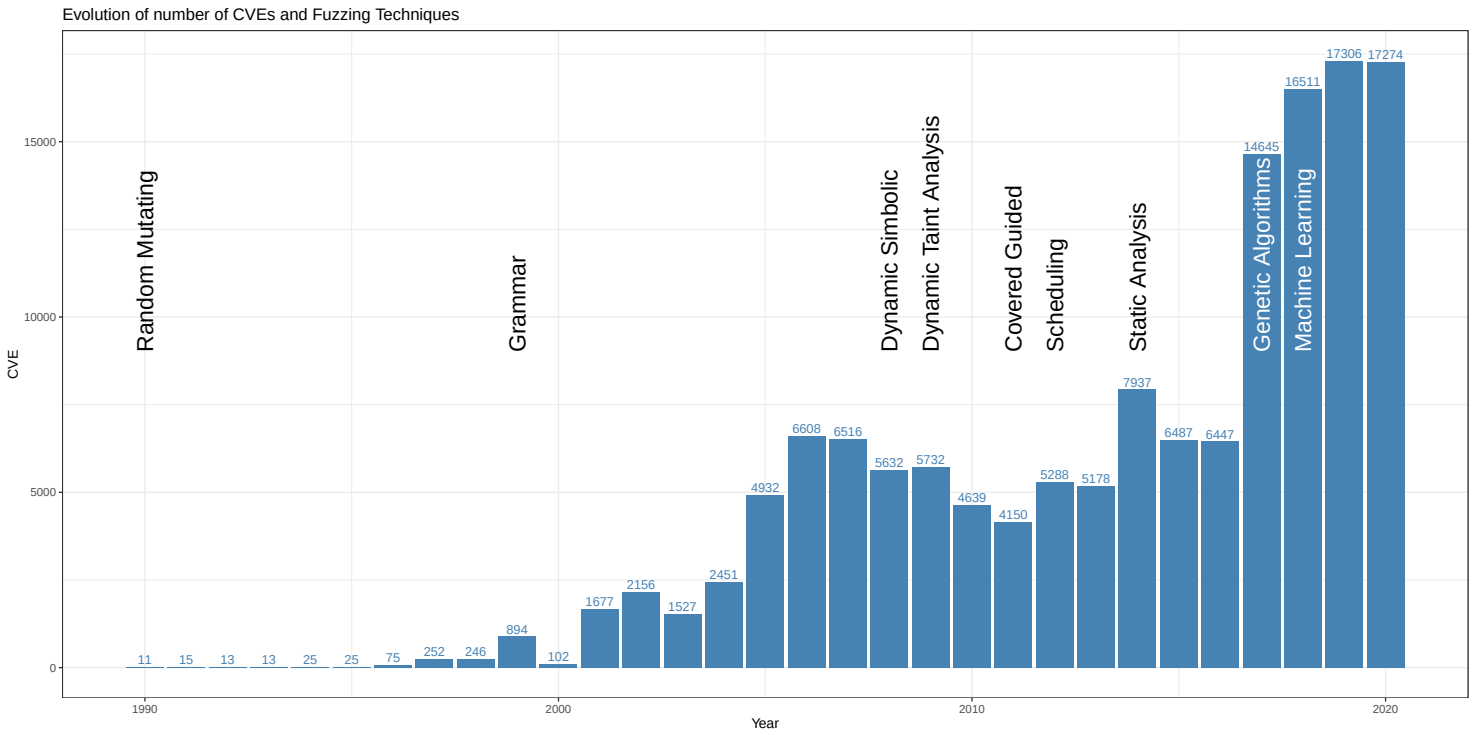

Fig. 4. Evolution of the adaptation of the different techniques to fuzzing

the search space. Some of the fuzzer that use grammars are Skyfire [49], GWF [24], or Peach [59] and they use them to fuzz complex SUTs.

- **Dynamic symbolic execution**. This is a black-box technique for generating fuzz data without intelligence but with a strategy that leverages the results obtained by symbolic execution to identify data value ranges for creating fuzz data. In this technique an interpreter follows the execution of the SUT from the input as a normal execution but without requiring a value to the end of the execution. Every time the interpreter reaches a control statement forks the symbolic execution storing the logical expression in terms of input variables. At the end of the execution of the different forks, a set of logical expressions in terms of input variables are obtained and the range of valid values are obtained from each logical expression. These ranges are used to seed the fuzz data generation and explore the SUT. As a result a significant improvement in the performance is obtained compared to other approaches. The process of symbolic execution is described in Fig. 5 comparing it with concrete execution. Although this technique has a very significant potential for exploring different paths it also has several drawbacks. Firstly, the explosion of potential paths limits the use in complex programs. Secondly, the scalability of the technique is also limited due to path explosion and due to the negative impact in memory consumption. Finally, the environment interactions when performing system calls may arise when execution reaches components that are no under control of the interpreter [73], [74], [75]. This technique is used in the context of fuzzing for facilitating the determination of the initial seed by leveraging the range of values provided by the symbolic execution, the range value data for specific sections [68], or for increasing the code coverage. This approach is commonly used in the white-box fuzzers [76], and KLEE [77] and SAGE [52] are two fuzzers that use it .

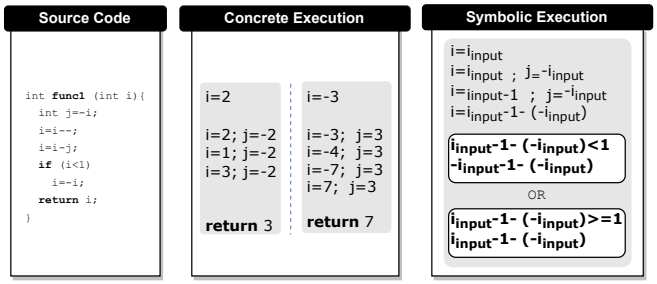

Fig. 5. Comparison between concrete and symbolic execution

- **Dynamic taint analysis**. It is a black-box technique based on checking variables modified by the user (registers and memory contents) throughout the execution of the SUT when a crash occurs. The process is depicted in Figure 6. The dynamic nature of the taint analysis requires a binary instrumentation framework with the aim of adding a pre/post handler on each instruction. Thus, it is possible to retrieve all the information about the instruction or the environment (memory) revealing useful information regarding the SUT. This information is very valuable and it is used for generating new fuzz data [24]. The main advantage of dynamic tainting is the improvement in the efficiency of the fuzzer. However, it has two major

drawbacks: under-tainting and over-tainting. The first one indicates that there are contaminated data that have not been marked as such. In contrast, over-tainting marks too many values as contaminated, as a result the false positive rate is increased. Some examples of this approach are: Vuzzer [78], and SYMFUZZ [79] use the dynamic taint analysis to find the input seed to mutate.

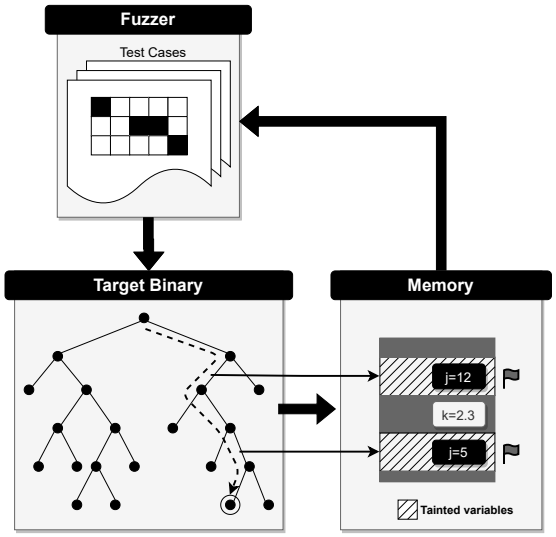

Fig. 6. Fuzzing with dynamic taint analysis

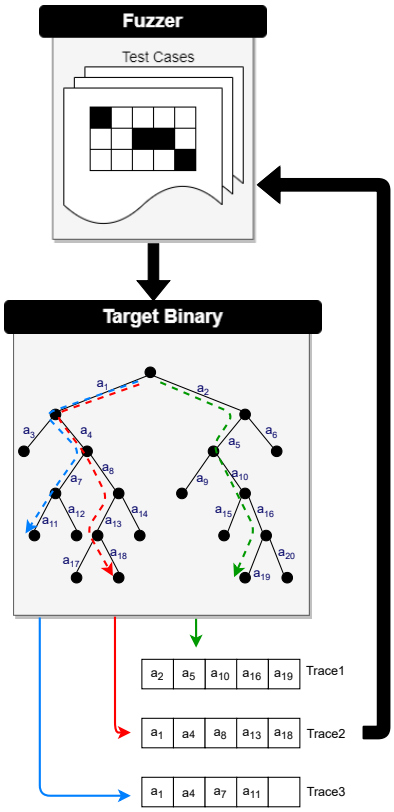

Fig. 7. Process of fuzzing with guided coverage technique

- **Guided covered**. It is a grey-box fuzzing technique that is based on program instrumentation to trace the execution of the SUT identifying code sections reached by the input provided. This information is used by the fuzzer to make informed decisions to select which inputs to mutate in the SUT [75], [80] as shown in Fig. 7. The main goal of this technique is to increase the code coverage [15] with the minimum test cases necessary [11]. Although this technique has a good performance a priori, it is governed by the complexity of the SUT, this is the main enemy of this technique, the higher complexity the lower performance [73], [81]. Some of the representative fuzz tools which use this technique are kALF [60] and Driller [82].
- **Scheduling algorithms**. It is a grey-box technique based on a lightweight instrumentation of the SUT which provides information of the memory. This approach is divided in two stages: exploration stage and exploitation stage. In exploration stage the contents of the memory is partially tainted to observe input values in memory when specific instructions such as compare instructions are executed. This inspection of memory creates association of input values with memory states known as *configuration* allowing to perform educated guesses with regard to which values to replace. Exploitation stage takes advantage of the previous results and exploit these to seed the fuzzer [11]. Examples of this kind of technique are BFF [83] and FuzzSim [24].
- **Static analysis**. It is a white-box technique that leverages the results of the static analysis of the source code to generate fuzz data with the aim of exploit the weaknesses located by the analysis. The main advantage is its simplicity, however, the false positive rate is high and the accuracy is low [73]. Some examples this kind of fuzzers are Dowser [53] and BORG [84].
- **Genetic algorithms**. This technique can be a black-box, grey-box, or white-box depending on the information used for taking decisions. It is a search-based method optimization technique based on the natural selection aimed to find optimal solutions to difficult optimization problems. In the area of fuzzing, the problem of finding vulnerabilities can be seen as a optimization problem where each candidate solution or individual represents a candidate fuzz data and the fitness function aims to represent the distance to a vulnerability in the SUT by including information of different nature such as dynamic information like warnings, execution timeouts, errors, crashes, or others, and static information like cyclomatic complexity among others [85], [86]. This technique represents a qualitative leap for searching new vulnerabilities but this requires a good estimate of the SUT behaviour. The process is represented in Fig. 8. AFL, one of the most well-known fuzzers today uses genetic algorithms to improve test generation [87].
- **Machine learning** This is not a technique, but rather a set of techniques that can play the role of black, grey or white-box depending on the modelling of the problem to solve. Machine learning is a set of data analytic techniques that allow computers to learn from data like humans and/or animals [55]. This kind of algorithms

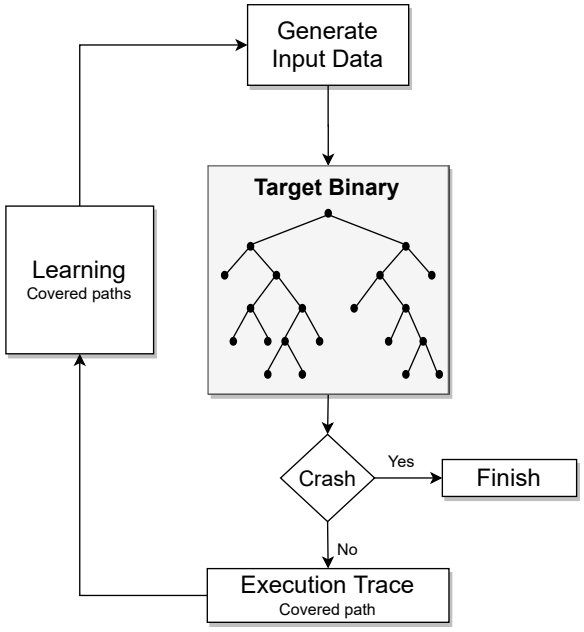

Fig. 8. Process to fuzz using evolutionary algorithms

utilizes computational/mathematical models to acquire knowledge from data without human expert intervention. The algorithms in this field base their performance on the quality and quantity of samples that describe the problem. A specific subset of models in this field is the deep learning very focused in complex and heterogeneous neural networks. Regarding fuzzing, this set of techniques have been applied to support different stages of the process of fuzzing such as seed generation, pre-processing of source/binary code, test case generation, fitness function, mutation operator, exploitability analysis, etc. [73]. In addition to this, existing metrics have been used to assess the performance. This is the most complete subset of techniques with successful results. The introduction of this kind of techniques is relatively new and a very good source of new probed techniques in this field. However, like the rest of the techniques there is a fundamental drawback regarding the need of enough and representative samples. An example of a fuzzer that use this kind of technique is Learn & Fuzz [63].

Table II summarizes the advantages and disadvantages of the different fuzzing techniques covered in this section.

IV. Fuzzing Embedded Systems

Fuzzers have evolved significantly since the creation of the first fuzzer, demonstrating its effectiveness finding vulnerabilities, although its use in embedded systems is not widespread. Hence, it is important to analyze the reasons that limit the use and the main characteristics that fuzzers should fulfill in order to be applied to embedded systems.

When considering fuzzers, a wide range of different alternatives exist. However, one of the most popular fuzzers is American Fuzzy Lop (AFL), which is widely used in academic and industrial areas [54]. AFL is centered on generating malformed input files to running processes (e.g. a PDF file to a document reader). Moreover, the development of many other fuzzers is based on AFL, such as, AFLFast [88], AFLGo [89], Skyfire [49], VUzzer [78], Steelix [90] or Angora [91]. Based on the popularity of AFL and its descendants, many of the newest fuzzers are compared with it. In addition, other popular fuzz tools covering other areas and also used for comparison are Honggfuzz [92], Peach [59], KLEE [77], SAGE [52] or Radamsa [93]. Although all of them are fuzzers, they have a different nature, making them more suitable to apply in different environments, particularly based on the input data they create (files, network traffic etc.).

As covered in Section II, finding vulnerabilities in embedded systems is a challenging endeavour, as the most restricted embedded systems do not have additional security mechanisms in place. Therefor, using fuzzing to detect vulnerabilities is an interesting option, as it can run outside the system and it analyzes the outputs of the SUT to find these vulnerabilities. Even so, it should be noted that in IT systems have security mechanisms pin place, such as MMU or MPU, that help to detect errors. As mentioned earlier, this is not always the situation in embedded systems. In addition, it must be considered that when an embedded system with limited resources freezes, it will probably stop responding. In these cases, the fuzzer will not have new data for the generation of the next inputs, decreasing its effectiveness. For that reason, it is necessary to find and leverage all possible methods to monitor the SUT.

With all its limitations, embedded system fuzzing has already shown some promising results in previous proofs of concepts, such as fuzzing SMS messages [25] and the Global System for Mobile communications (GSM) protocol [26] to detect vulnerabilities in smartphones, fuzzing credit cards to exceed its limit [28] and also fuzzing the Controller Area Network (CAN) protocol of cars to unlock their doors or disabling their lights [27].

A. Measurable characteristics in embedded fuzzing

As out-of-the-box fuzzing for embedded systems is not viable for vulnerability and crash discovery, it is necessary to focus on characteristics present in embedded systems that could potentially be leveraged for vulnerability detection. These characteristics can be grouped in four main categories: fault types, response time, waiting time and physical response nature.

- **Fault type**. This category groups all kinds of errors related to the memory and numerical computation that can derive in a different behaviour compared to general-purpose IT systems.
  - **Stack overflow** takes place when a program writes to a memory address in the program call stack out of the designated data structure. The usual consequence of the stack overflow is a crash. In addition, embedded systems with fewer resources can have problems detecting these errors. In some cases, the error is detected later when the system stops responding to requests or it may not have any effect, so if the memory is not corrupted it will not have any visible effect. In more advanced

TABLE II
SUMMARY OF THE ADVANTAGES AND DISADVANTAGES OF DIFFERENT TECHNIQUES ADDED TO FUZZING

Technique	Advantages	Disadvantages
Random Mutation	High implementation speed and scalability	Many invalid inputs
Grammar Representation	Complex structure	Not completely automatic
Dynamic Symbolic Execution	Runs the code to monitor	Limited path explosion, environment interactions
Dynamic Taint Analysis	Reaches specific functions	Under-tainting and over-tainting
Guided-Covered	Scans many inputs efficiently	Difficulties with complex checks
Scheduling Algorithm	Improves black-box testing	Problems in embedded devices
Static Analysis	Not executed, approximate behaviour	False positives and negatives
Genetic Algorithms	Improve new generations	Result depends on stop criteria
Machine Learning	Intelligent fuzzing	Needs large data quantities

embedded systems with an OS, errors can be detected as the execution is stopped and it is usually warned by a message or a signal [18].

– **Segmentation fault** appears when it is not allowed access to specific region of the memory from the main process. When the system throws an exception of this class is providing clues that there is unauthorized access to memory. However, in the worst case, when dealing with systems with no OS, this event usually has no effect [18], and therefore there are no observable event, in embedded systems with an OS or embedded OS are usually detected by an observable crash or reboot but it is not reasonable to assume the detection of this kind of events.
– **Memory corruption** takes place when the memory is modified without an explicit assignment. It can cause many problems such as invalid pointer values, incorrect data or a crash of the system. On the embedded devices the memory corruptions are less visible, what causes a decrease on the effectiveness of the dynamic testing techniques, so the silent memory corruptions are habitual on this devices. All derived executions from memory corruptions can be categorized in four types of classes. Firstly, the system can stop responding to requests or it can have a late crash. Secondly, the system will start to malfunction. Thirdly, it may continue to function in the same way and the error will not have any effect. Fourthly, in systems with OS, which are usually the most resourceful embedded systems, the reaction to this type of error is usually an observable crash [18].
– **Numerical errors**. In addition to the errors mentioned above, on one hand the absence of managing typical over/under-flow errors can take place without any visible effect. On the other hand, the limitations of embedded systems such as the absence of subset of instructions for computing division may involve the use of supplementary numerical computation libraries. As a consequence in the most constrained devices a numerical error can derive into a pure software error that like in the first case can take place without any visible effect or worse these can derive into a memory corruption.

- **Response Time**. This is the elapsed time from the reception of fuzz data until the data is finally processed and ready to be sent back as it shows Fig. 9. Embedded systems need much more time to respond, due to the resources limitation as a consequence fuzzers must be adapted to this increase of time, otherwise the rate of false negative can increase. The response time is therefore a measure that impact directly on the performance [94].

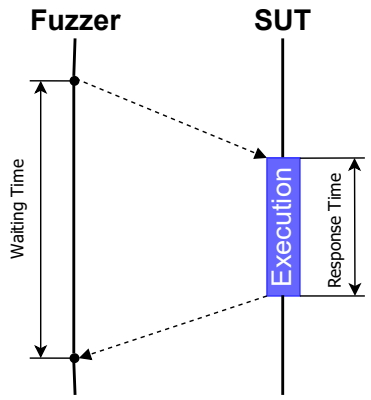

Fig. 9. Response time and waiting time of the fuzzing process

- **Waiting Time.** It is the period of time that elapsed from the request is launched to the SUT until its response arrives. It is necessary to consider the possibility that the answer could not arrive (e.g. the system is non-responsive or has crashed). In such a case the fuzzer has to decide whether to send another test case or to stop testing because the device has died or it is taking too much time to run. It can also happen that the time between tests is less than the waiting time, in this case, it is necessary to be careful, as responses can come mixed as happens in Fig 10. In the case of embedded systems it will be necessary to increase the waiting time of the fuzzer, since if it is not adapted, it can detect an error in the system when it is working correctly, causing an increase in false positives.
- **Physical response nature**. The physical response of a system in different situations can give valuable information about what is occurring to it. In contrast to general-purpose IT systems, embedded systems are small and suffer physically from increasing the temperature of the device when the volume and the rate of fuzz data increases, they tend to get warmer at higher loads. Moreover, in this kind of systems the load on the system falls only on the CPU. To prevent this, it is necessary to measure temperature during the fuzz testing and observe

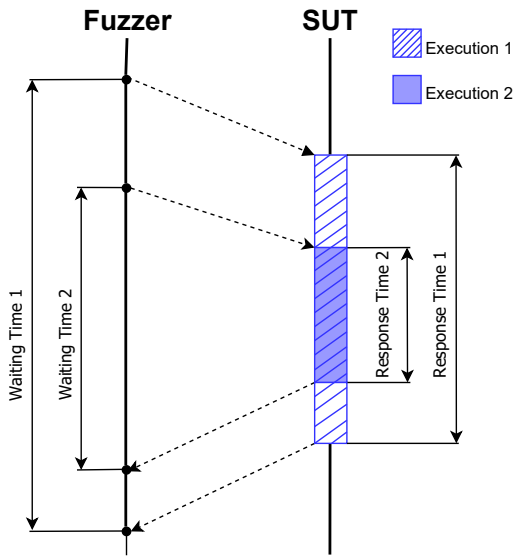

Fig. 10. Tests executions with less time between tests than the waiting time

that the temperature does not reach undesired values. The temperature affects to the reliability of the system, but can also affect to the operation power and cost [95]. Therefore, it can be a helpful factor to determine if there have been any changes in the CPU.

B. Enhanced observation techniques

In order to circumvent issues related to poor monitoring and observation abilities when fuzzing embedded software, several approaches have been proposed to make crashes and errors observable. These enhanced observation techniques are based on the use of emulation, source code analysis and sanitization.

- **Full emulation**. Costin et al. [96] and Chen et al. [97] have shown that under specific conditions, applications extracted can be executed inside a generic system running on a off-the-shelf emulator. Based on this emulator, it is possible to collect information regarding the status of the memory. However, the main disadvantage of this approach is that it is not possible to test peripheral activity. That is, when peripheral use is necessary (e.g. to test code interacting with these peripherals), these are not accessible to the fuzzer, as they are neither present in the target nor emulated.
- **Partial emulation**. In order to overcome the drawbacks of full emulation, some authors have extended the emulation from systems without an operating systems to those with an operating system [98] where only peripheral are excluded from emulation and all instructions related to the peripherals are redirected to real external peripherals. Based on this approach Muench et al. [18] obtain significant results in detecting classical memory related errors. However, such an approach suffers from a significant false positive and negative rates [46].
- **Source Code Availability.** It has proven helpful to increase fuzzing performance, particularly aiding exploration. However, this availability is not always possible, especially when auditing third-party software.

There are a subset of specific techniques related to the assessment of the memory that can guide the fuzzer. These techniques are based on the concept of *sanitization*. A sanitizer instruments the applications by inserting checking instructions in order to monitor all read and write operations to/from the memory. In this sense, two kinds of sanitizers exist:

- **Dynamic Binary Sanitizers.** This technique allows to instrument the application at run time. However, dynamic sanitizers have a significant drawback in embedded devices, as they cause a significant performance overhead and require special software/hardware functionalities that are not widely available in embedded systems. Due to these reasons, dynamic sanitization adoption is limited [99].
- **Static Binary Sanitizers**. First presented by Salehi et al. [46]. , static sanitizers have lower overhead than their dynamic counterparts and they also yield better results when compared to proposals based on partial emulation. Static binary santization is a multi-stage process (depicted in Fig. 11):
  - ***Stage 1. Static disassembly***. This is the process of parsing the executable region of the input binary file and decoding the content into their a human readable format.

  The process of instrumentation is typically divided into two steps. The first one is responsible for locating every relevant instruction of the code that needs to be instrumented whereas the second one is responsible for inserting the instrumentation code. In this case, the process is much more sophisticated and structured, and it is divided into four steps: memory instruction extraction, specification generation, binary instrumentation and mapping generation.
  - ***Stage 2. Memory instruction extraction***. The goal of this stage is to identify which instructions are relevant for the next stage: the specification generation.
  - ***Stage 3. Specification generation***. This process is in charge of creating the specification based on the information obtained from the extracted instructions.
  - ***Stage 4. Binary instrumentation and mapping generation***. Based on the specification generated in the previous stage, this process performs the instrumentation of the assembler code. The new segments of code perform two tasks:
    - Creation of a metadata structure in memory where all out-of-bounds memory regions are controlled by means of storing of the boundary address.
    - The access control to out-of-bound memory addresses by consulting the metadata structure.
  - ***Stage 5. Reassembling***. Based on the previous instrumentation, the source code is re-assembled in order to create a new binary.

  Once this process has finished and starts executing, a specific region in memory is created called the *metadata region* which contains the boundary address of every memory value facilitating the control of any access to

address out of the bounds. As a result invisible errors become observable. All this process has been depicted in Figure 11. This technique has reduced the false and positive errors obtained with previous techniques without requiring a full/partial emulation.

Although these mechanisms partially circumvent the difficulties posed by embedded systems, it is still necessary to increase the performance of fuzzers by different reasons. Firstly, it is necessary to reduce the false positive rate, the response time, the waiting time and the fault type can affect negatively to this rate. Secondly, it is necessary to reduce the false negative rate by improving the detection of some type of fault types. Finally, an improvement of the test rate, it is necessary an improvement in the capability of generating more test data while maintaining the false positive and negative rate low without manipulating or modifying the binary or the source code and without overflowing the embedded system.

V. Discussion

In this section we will provide a review of the existing fuzzing techniques summarized in Table II, as well as a review of the features of these techniques that are required in embedded systems summarized in Table III. A discussion on the evolution and motivation of these techniques will be provided with a special focus on the application in embedded systems.

A. Fuzzing Trends

A fuzzer is a method that aims to find vulnerabilities by sending specially crafted data and analyzing the feedback received. The first fuzzers did not take into account the feedback of the SUT to generate the data, the only relevant information used was the successful findings of new vulnerabilities. As a consequence the results were very poor compared to any recent approach. The next leap in evolution was to consider a proper mechanism to generate crafted inputs but this was still without considering the feedback, this change only represented a small improvement not a major one.

The next great qualitative leap in evolution of the fuzzers begins by considering the information provided from the SUT in order to improve aspects such as the generation of crafted data. At this point, the fuzzers improve the capability to detect vulnerabilities. This represented a clear division between smart and dumb fuzzers where no information is used. The Table III shows that the majority of the fuzzers are smart and few of them are dumb.

From this point, the nature of the information considered to improve the fuzzer makes the difference between the two main families of fuzzers: black-box and white-box. The first family ignores the information based on the source code, considering only the binaries in the best case. Whereas the second family takes into account the information based on the source code. The source code is instrumented to provide additional information jointly with response of the SUT when the fuzzer is operating.

A white-box strategy represents the most used techniques because this provides the maximum possible information. However, this is not always possible which limits the use in specific situations. As a consequence, some fuzzers do not make use of the complete information but they use partial information and the instrumentation to operate. Table III shows that the proportion of strategies is close to one third.

The use of information used represented a clear difference between the different families of fuzzers. Now, the next qualitative leap was in the technique that best makes use of this information according to the family. The initial techniques made use of the instrumentation whereas the last ones are based on evolutionary approaches that can take advantage of different sources of information. In this sense, there has been a displacement in the last years from less flexible techniques to the most modern approaches based on machine learning. Consequently, the use of machine learning represents a clear advantage in any of the approaches.

B. Embedded Fuzzing

The evolution of fuzzers represents a priori a significant and promising technique for finding weaknesses in embedded systems. However, the nature of these systems imposes several constraints that limit the use of a great number of the fuzzers as we will show. In this section we will show a different taxonomy of fuzzers according to a set of more suitable features for embedded systems such as: Source Code, Fault Types, Instrumentation, Support of the operating system, and the Target (see Table IV) as well as an analysis of the impact of these features.

Firstly, the availability of the source code in embedded systems is not always possible. In many cases, the manufacturer imposes their software developing kit for developing applications limiting the possibilities of using tools. In addition to this, according to the standard "IEC 62443-4 Practice 5, SVV-3 Vulnerability testing", the application of fuzzers should be carried out in external interfaces without additional information. This reduces possible candidate fuzzer numbers by 25%, that is, decreasing the number of eligible fuzzers from 41 to 31.

Secondly, the capability of detection of fault types plays a significant role in embedded systems as we have depicted previously in Section IV. The behaviour of an embedded system when a failure takes place does not correspond to the expected behaviour as in standard systems. In embedded systems, a failure can involve that the system can continue operating with wrong values because there is no a Memory Management Unit or the system can stop completely. As a consequence the number of fuzzers that are able to detect different kind of failures is very low.

Thirdly, as a consequence of the first point, the possibility of instrumenting the source code is not possible due to the same constraints but in very constrained embedded systems there is no user interface, or shell to interact which makes impossible any collection. This involves that is necessary to discard 17 additional fuzzers representing a reduction of a 65% of the total.

Fourthly, embedded systems can operate with/without the support of an operating system, so it is not always possible to

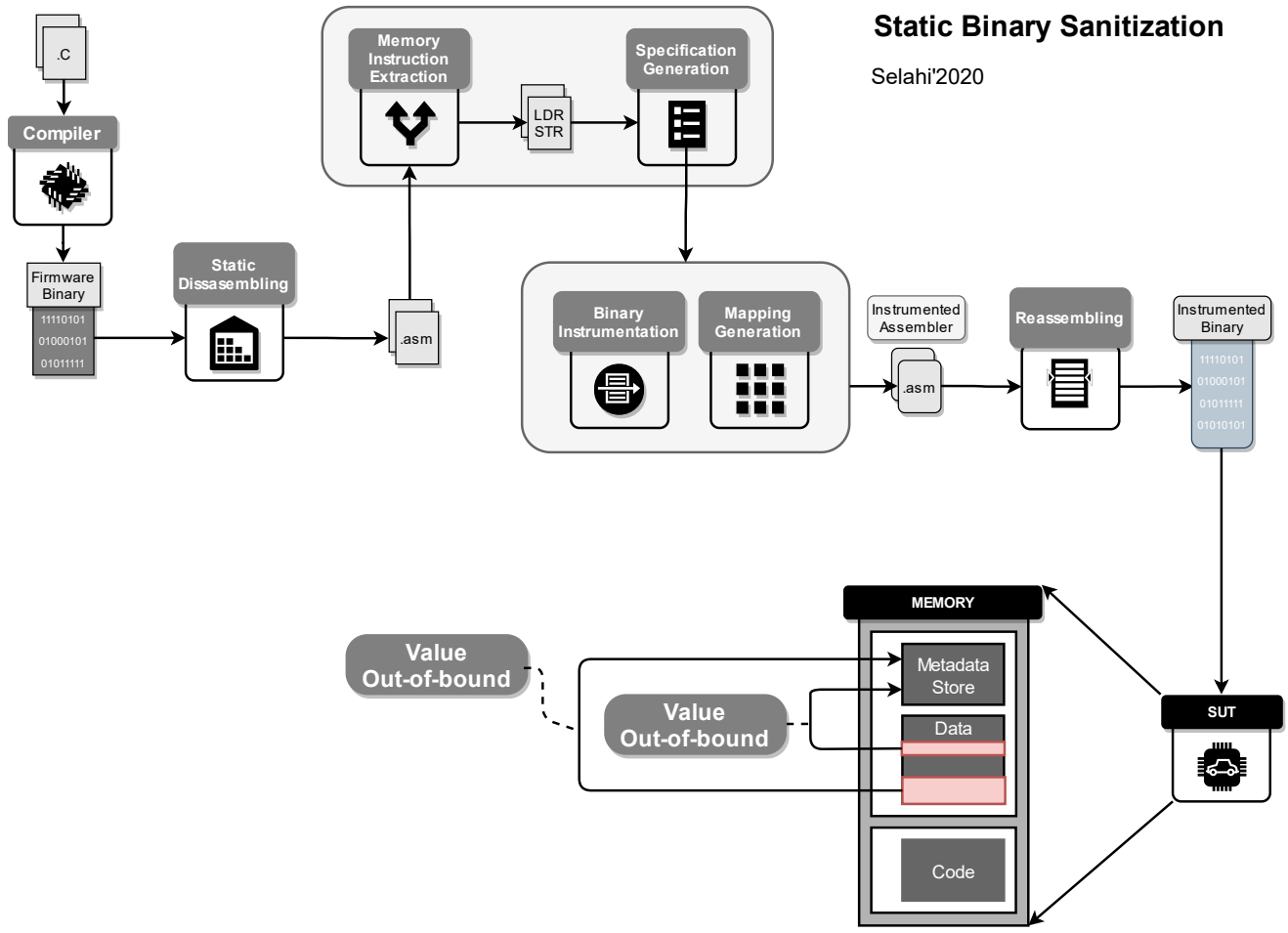

Fig. 11. Static Binary Sanitization for embedded systems

assume the support of this to carry out the fuzzing process. This may be the primary factor for discarding many fuzzer techniques reducing the remaining 14 fuzzers to 4 fuzzers.

Finally, in contrast to classical fuzzers, the target in embedded systems is much more limited than general-purpose systems. In some embedded systems makes no sense to target file-systems because there is no such possibility, and therefore it is not realistic to assume this. As a consequence, all remaining fuzzers focused in this class of targets that involves the presence of a file-system should be discarded.

In summary, the mere fact of imposing some limitations such as the use of source code, or the instrumentation has a significant impact on the availability of fuzzers. If we add additional constraints such as the participation of the operating system depicts an overview where is necessary to invest more resources to create specific fuzzers for embedded systems.

C. Embedded System Fuzzing Qualities

Previous sections has shown a broad range of fuzzing techniques but also the challenge that poses to choose/design a suitable fuzzer for embedded systems when specific features are taken into account in contrast to general-purpose systems.

Embedded systems impose some constraints, but the most important factor is the limitation of feedback information from the embedded system. As we have explained previously, the information from the SUT is essential to find weaknesses in the source code, and it is precisely the lack of information what characterises the nature of embedded systems. The difficulties for obtaining or instrumenting source code involves that it is not possible to obtain more information from the SUT requiring the use of black-box approaches, that is, the most difficult approach.

As a consequence, a suitable fuzzer for embedded systems should compensate this lack of information with more intelligence jointly with the most advanced techniques as primary requirements. The fuzzer should only deal with the response of the SUT without taking into account the source code/instrumentation, that is, without any additional help. This involves that it is necessary to use advanced techniques such as those based on machine learning for learning from the all available information.

In addition to this, it is also relevant and significant the consideration of the behavior of each type of failure. A specific type of failure in a embedded system does not behave in the same way as in general-purpose system which can impact in the operation in two aspects. Firstly, the first situation is when a failure is not detected and the fuzzer continues operating normally, this involves a false negative. The second situation

TABLE III
COMPARISON OF FUZZ TOOLS TAKING ACCOUNT THE FEATURES OF FUZZERS TO DISTINGUISH EACH OTHER

Methods	Type			Input Generation		Intelligence		Exploration Strategy		Techniques									
	1	2	3	1	2	1	2	1	2	1	2	3	4	5	6	7	8	9
	1. White-Box 2. Grey-Box 3. Black-Box			1. Mutation 2. Generation		1. Smart 2. Dumb		1. Covered 2. Directed		1. Random Mutating 2. Grammar Representation 3. Dynamic Symbolic Execution 4. Dynamic Taint Analysis 5. Coverage Guided 6. Scheduling Algorithms 7. Static Analysis 8. Genetic Algorithms 9. Machine Learning								
AFL [16], [100]	□	■	□	■	□	■	□	■	□	□	□	□	■	□	□	■	□	
AFLFast [66], [88], [89]	□	■	□	■	□	■	□	■	□	□	□	□	■	■	□	□	□	
AFLGo [66]	□	■	□	■	□	■	□	□	■	□	□	□	■	■	□	□	□	
Angora [57], [101]	□	■	□	■	□	■	□	□	■	□	□	□	■	■	□	□	□	□
BFF [83], [102]	□	□	■	■	□	□	■	□	□	■	□	□	□	□	■	□	□	□
BORG [84]	□	□	■	■	□	■	□	□	□	□	□	■	■	□	□	■	□	□
BUZZFUZZ [103]	■	□	□	□	■	■	□	□	■	■	□	□	■	□	□	□	□	□
Dowser [53], [70]	■	□	□	□	■	■	□	□	■	□	□	■	■	□	□	■	□	□
DrE [104], [105]	■	□	□	□	■	■	□	□	■	□	□	■	□	□	□	■	□	□
Driller [82]	■	□	□	■	□	□	■	□	■	□	□	■	□	□	□	□	□	□
FIE [106]	■	□	□	□	■	■	□	■	□	□	□	■	□	□	□	□	□	□
Frankenstein [107]	□	■	□	■	□	■	□	□	□	■	□	□	□	■	□	□	□	□
fuzz [13]	□	□	■	■	□	□	■	□	□	■	□	□	□	□	□	□	□	□
FuzzSim [83]	□	□	■	■	□	□	■	□	□	■	□	□	□	□	■	□	□	□
GWF [50]	■	□	□	□	■	□	■	■	□	■	■	□	□	□	□	□	□	□
Honggfuzz [92]	□	■	□	■	□	■	□	□	■	□	□	□	□	■	□	□	□	□
IoTFuzzer [64]	□	□	■	■	□	■	□	□	□	□	□	■	□	□	□	□	□	□
kALF [60]	□	■	□	■	□	■	□	■	□	□	□	□	□	■	□	□	□	□
KLEE [77]	■	□	□	□	■	■	□	□	■	□	□	■	□	□	□	□	□	□
Learn&Fuzz [63]	□	■	□	□	■	■	□	□	■	□	□	□	□	□	□	□	□	■
libfuzzer [11], [108], [109]	□	■	□	■	□	■	□	□	■	□	□	□	□	■	□	□	□	□
MoWF [110]	■	□	□	□	■	■	□	□	■	■	■	■	□	□	□	□	□	□
PAFL [111]	□	■	□	■	□	■	□	□	■	□	□	□	□	■	□	□	□	□
Peach [17], [59], [112]	□	□	■	■	■	■	□	□	□	■	■	□	□	□	□	□	□	□
QuickFuzz [113]	□	□	■	■	□	□	■	□	□	■	□	□	□	□	□	□	□	□
Radamsa [93]	□	□	■	■	□	■	□	□	□	■	□	□	□	□	□	□	□	□
RedQueen [11], [114]	□	■	□	■	□	■	□	□	■	□	□	■	□	□	□	□	□	□
S2E [115]	□	□	■	□	■	■	□	□	□	□	□	■	□	□	□	□	□	□
SAGE [52], [116]	■	□	□	□	■	■	□	■	□	□	□	■	□	■	□	□	□	□
Skyfire [49]	□	■	□	□	■	■	□	□	■	□	■	□	□	■	□	□	□	□
SLF [117]	□	■	□	■	□	■	□	□	■	■	□	■	□	□	□	□	□	□
SmartFuzz [65]	■	□	□	■	□	■	□	□	■	□	□	■	□	■	□	□	□	□
SPIKE [17], [22]	□	□	■	□	■	■	□	□	□	■	■	□	□	□	□	□	□	□
Steelix [90]	□	■	□	■	□	■	□	□	■	□	□	□	□	■	□	■	□	□
Sulley [58], [118]	□	□	■	■	■	■	□	□	□	■	■	□	□	□	□	□	□	□
SYMFUZZ [72]	■	□	■	■	□	■	□	□	□	■	□	□	■	□	■	□	□	□
Syzkaller [119], [120]	□	■	□	■	■	■	□	■	□	□	□	□	□	■	□	□	□	□
TaintScope [121]	□	■	□	■	□	■	□	□	■	□	□	■	■	□	□	□	□	□
T-Fuzz [80], [122]	□	■	□	■	□	■	□	□	■	□	□	■	□	■	□	□	□	□
VUzzer [78]	□	■	□	■	□	■	□	■	□	□	□	□	■	■	□	■	□	□
zzuf [65]	□	□	■	■	□	□	■	□	□	■	□	□	□	□	□	□	□	□

is when the failure is detected much more later, and the fuzzer learns from incorrect information, generating incorrect inputs. As a conclusion, the fuzzer should be able to detect at least some of these situations and learn correctly from the original input and not from later inputs.

Finally, many approaches require the intervention of the OS to support the fuzzer, in the case of embedded systems, the fuzzer should operate without any help from the operating system, the limitations of some embedded systems make this not recommendable. Thus, the design of a fuzzer specialized for embedded systems should not be based on the operating system of the SUT.

VI. FUTURE DIRECTIONS

In this paper different fuzzing tools have been categorized and classified showing the relevance and usefulness of the fuzzing approach to find weakness in software with independence of the target and the strategy used. However, this survey has also shown several flaws in the design of fuzzers when applied to embedded systems as well as in the methods used for measuring the performance of these. Therefore, there are several open issues that could lead to future research directions in the field of fuzzing and IoT and embedded devices. We have grouped such directions in two main categories: fuzzer evaluation, enhanced embedded fuzzing architecture, and IoT and embedded fuzzing algorithms.

TABLE IV
COMPARISON OF FUZZERS CONSIDERING NECESSARY FEATURES TO FUZZ EMBEDDED SYSTEMS

Methods	Source Code		Fault Type				Work with OS			Instrumentation			Target Support								
	1	2	1	2	3	4	1	2	3	1	2	3	1	2	3	4	5	6	7	8	9
	1. Yes 2. No		1. Stack Overflow 2. Segmentation Fault 3. Memory Errors 4. Other				1. Yes 2. No 3. Not specified			1. Yes 2. No 3. Not specified			1. File 2. Library 3. Firmware 4. OS Kernel 5. Network Protocol 6. Applications 7. General-Purpose 8. Embedded Systems 9. Web browser								
AFL [16], [100]	□	■	■	□	■	□	■	□	□	■	□	□	■	□	□	□	□	■	■	□	□
AFLFast [66], [88], [89]	□	■	■	■	■	□	■	□	□	■	□	□	■	□	□	□	□	□	■	□	□
AFLGo [66]	■	□	■	□	■	■	■	□	□	■	□	□	■	□	□	□	□	□	■	□	□
Angora [57], [101]	■	□	■	□	□	■	□	■	□	■	□	□	■	□	□	□	□	□	■	□	□
BFF [83], [102]	□	■	□	□	□	■	■	□	□	□	■	□	■	□	□	□	□	□	■	□	□
BORG [84]	□	■	□	□	■	■	■	□	□	□	■	□	□	■	□	□	□	■	■	□	□
BUZZFUZZ [103]	■	□	□	□	■	■	□	■	□	■	□	□	■	□	□	□	□	□	■	□	□
Dowser [53], [70]	■	□	■	□	□	□	□	■	□	□	□	■	□	□	□	□	□	■	■	□	□
DrE [104], [105]	■	□	□	□	■	□	□	□	■	□	□	■	□	□	■	□	□	□	□	■	□
Driller [82]	□	■	■	□	■	□	■	□	□	■	□	□	■	□	□	□	□	□	■	□	□
FIE [106]	■	□	□	□	■	■	■	□	□	□	□	■	■	□	□	□	□	□	□	■	□
Frankenstein [107]	□	■	□	□	■	□	■	□	□	□	□	□	■	□	□	□	□	□	■	□	□
fuzz [13]	□	■	■	□	□	□	■	□	□	□	■	□	□	□	□	□	□	■	■	□	□
FuzzSim [83]	□	■	□	□	□	■	■	□	□	□	■	□	■	□	□	□	□	□	■	□	□
GWF [50]	□	■	■	□	■	■	□	□	■	■	□	□	■	□	□	□	□	□	■	□	□
Honggfuzz [92]	□	■	■	□	■	■	■	□	□	■	□	□	■	□	□	□	□	□	■	□	□
IoTFuzzer [64]	□	■	□	□	□	■	□	■	□	□	■	□	□	□	■	□	□	□	□	■	□
kALF [60]	□	■	■	□	■	■	■	□	□	■	□	□	□	□	□	■	□	□	■	□	□
KLEE [77]	■	□	■	□	■	□	■	□	□	■	□	□	□	□	□	□	□	■	■	□	□
Learn&Fuzz [63]	□	■	■	□	□	□	■	□	□	□	■	□	■	□	□	□	□	□	■	□	□
libfuzzer [11], [108], [109]	■	□	□	□	□	■	■	□	□	■	□	□	□	■	□	□	□	□	■	□	□
MoWF [110]	□	■	■	□	□	■	■	□	□	□	□	■	■	□	□	□	□	□	■	□	□
PAFL [111]	□	■	□	■	□	■	□	■	□	□	□	■	□	□	□	□	□	■	■	□	□
Peach [17], [59], [112]	□	■	■	■	■	■	■	□	□	□	■	□	■	□	□	□	■	□	■	■	□
QuickFuzz [113]	□	■	■	■	□	□	■	□	□	□	■	□	□	□	□	□	□	■	■	□	□
Radamsa [93]	□	■	□	□	□	■	■	□	□	□	■	□	■	□	□	□	□	□	■	□	■
RedQueen [11], [114]	□	■	■	□	■	■	■	□	□	■	□	□	■	□	□	■	□	□	■	□	□
S2E [115]	□	■	□	■	■	□	■	□	□	□	■	□	□	□	□	□	□	□	□	■	□
SAGE [52], [116]	□	■	■	□	□	□	■	□	□	■	□	□	□	□	□	□	□	■	■	□	□
Skyfire [49]	■	□	■	□	■	■	■	□	□	□	□	■	■	□	□	□	□	□	■	□	□
SLF [117]	□	■	□	□	□	■	□	■	□	□	■	□	■	□	□	□	□	□	■	□	□
SmartFuzz [65]	□	■	■	■	■	■	□	■	□	□	■	□	□	□	□	□	□	■	■	□	□
SPIKE [11]	□	■	■	□	□	■	■	□	□	□	■	□	□	□	□	□	■	□	■	□	□
Steelix [90]	□	■	□	□	□	■	□	□	■	■	□	□	□	□	□	□	□	■	■	□	□
Sulley [58], [118]	□	■	■	□	□	□	■	□	□	■	□	□	■	□	□	□	■	□	■	□	□
SYMFUZZ [72]	□	■	■	□	□	■	■	□	□	■	□	□	■	□	□	□	□	□	■	□	□
Syzkaller [119], [120]	□	■	□	□	□	■	■	□	□	■	□	□	□	□	□	■	□	□	■	□	□
TaintScope [121]	□	■	■	□	■	■	■	□	□	■	□	□	□	□	□	□	□	■	■	□	□
T-Fuzz [80], [122]	■	□	■	□	□	□	□	■	□	■	□	□	□	□	□	□	■	□	■	□	□
VUzzer [78]	□	■	□	□	□	■	■	□	□	■	□	□	■	□	□	□	□	□	■	□	□
zzuf [65]	□	■	□	□	□	■	□	■	□	□	■	□	■	□	□	□	□	□	■	□	□

A. *Fuzzer evaluation*

This subsection covers the current challenges for accurate comparison of fuzzers in general and, particularly, in the embedded field.

1) Evaluation methodology: The survey shows that there is no a standardized or consensual methodology to perform an assessment of the performance of any fuzzing tool according to a predefined set of key performance indicators [79]. Even it is not an standard evaluation methodology, it is necessary to measure the performance of any fuzzer in terms of quality and quantity in the most objective and fair way. The quality of a fuzzer should be measured by using standardized metrics such as *Accuracy*, the *False Discovery Rate*, or other classical metrics found in statistics and machine learning that can reflect with more objectivity the quality. In this sense, only a small portion of the authors use subsets of these metrics to assess the performance and the majority of them use their own mechanisms in a specific context to show the advantages, so nowadays, it is not possible to compare the fuzzers only analyzing the literature. The quantity, on the other hand, it is also a key performance indicator specially relevant in embedded systems due to the limitations of these, and it should measure the capability to perform a number of valid fuzz tests in a period of time without exhausting the resources of the system. The combination of both set of key performance indicators can provide a more accurate view of the performance of any fuzz tool and this should be a future direction in the design of fuzzers.

Hence, the use of a clear and standardized methodology to measure the performance could represent a significant advance in measuring the performance of fuzzers, but it is no less important to have standardized test data to measure the performance. In this context, the test data should represent a set of well defined programs that implements known weaknesses. The availability of data sheets of this nature could help when comparing results, with standarize data it would be easier to compare the quality of different fuzzers. In addition to this, it is necessary to implement these vulnerabilities under different levels of complexity in terms of control structures. This could provide insights of the capacity of the fuzzer under different levels of complexity present in the real world, from low hanging fruits to complex vulnerabilities.

Therefore, before developing a new fuzzer algorithm it will be necessary to define a evaluation methodology specifying the metrics for measurement and the data to perform the evaluation. In addition, it is necessary to set the experimental conditions.

2) Means of evaluation: Data and program availability: One of the main issues when aiming to compare results from different embedded fuzzing techniques is the necessity of a common benchmark and resources that can aid in interpreting results and choosing the best alternative. Such resources can be grouped in three main types:

- Vulnerable SUT targets. This refers to targets that contain specific and/or undiscovered bugs and, when necessary, bugs present only in specific types of devices. In this sense, the scientific community have used as two kinds of data:
  - Artificially injected vulnerabilities in existing libraries and applications. The most popular example for this purpose is LAVA [123] that injects vulnerabilities in GNU/Linux applications.
  - Existing open source applications and libraries with known and unknown vulnerabilities. In this case, programs of different nature have been used ranging from classical tools in Linux (such as tcpdump [124], jasper [125], objdump [57], uniq,...) to NIST Juliet test suite [126] for C/C++ or even utilities present in embedded systems such as *mbedTLS* [127], [128] and *expat* [129], [18].

  While there are some tools at the disposal of the research community, there is still a wide range to cover: no LAVA-like framework [130] exists for embedded and IoT vulnerability addition and the public availability of embedded-only binaries with well known vulnerabilities (found or induced) is still scarce. The creation of such data sheets and binaries would greatly benefit the community as a whole.
- Fuzzers and tools source code. Source code availability is necessary to be able to build fuzzers on top of different platforms and architectures. This would ease result reproducibility across different environments. Therefore, it would be necessary to publish the source code of fuzzers, the required toolchains, as well as the instruction to configure, compile and run the fuzzers, instrumentation tools, emulation environments. A short list of available tools is as follows:
  - Algorithms: AFL [54], AFLFast [88], AFLGo [131], Angora [91], Driller [132], Frankestein [107], Honggfuzz [92], QuickFuzz [133], Radamsa [93], SmartFuzz [134], Syzkaller [119], Vuzzer [135]
  - Emulation environments: QEMU [82], [136], QEMU STM32 [137], Unicorn [138], [107], hal-fuzz [139], Surrogates [140]
  - Sanitizers [141]: AddressSanitizer [142], ThreadSanitizer [143], MemorySanitizer [144], UndefinedBehaviorSanitizer [145], DataFlowSanitizer [146], LeakSanitizer [147]
  - Binary Instrumentation: Valgrind [148], Pin [149], DynamoRio [150], Dyninst [151]

  Again, while researchers have a wide plethora of public tools at hand, it is necessary to enrich the embedded fuzzing ecosystem to further develop the field.
- Embedded device data sheets. Having public information about different embedded devices, their properties and specific models is of utmost importance to enable and compare any hardware-oriented research to perform any kind of comparison. There is no such information nowadays.

In general, while a basic shared knowledge exists for embedded system fuzzing, it is true that when compared to general-purpose systems, the absence of these aspects difficulties not only the development of new fuzzes but the fair comparison of existing and new approaches. Thus, it is extremely important the presence of such a repository.

B. Enhanced embedded fuzzing architecture

As stated previously, the main problem when dealing with fuzzing and embedded devices is the total or partial absence of observability. That is, when a failure takes place, embedded systems have low or non-existent capability to show that abnormal situation has happened.

With the aim of overcoming this problem, it is necessary to explore or to develop new mechanisms that allow detecting failures in embedded systems. Moreover, it is also necessary to focus in the improvement of other stages of the fuzzing process. With this idea in mind, a possible enhanced embedded fuzzing architecture (shown in Fig. 12) is proposed with the following modules:

- **Generator**. This mechanism is responsible for generating new fuzzed data but in contrast with traditional generators the generator should be both SUT-independent and agnostic. As a consequence, this opens the door for integrating new approaches from other fields such as those from artificial intelligence or advanced statistics among others.
- **Learning from all collected data**. New, enhanced fuzzers should be adapted to deal with data of different nature to take advantage of all information and explore the space of potential vulnerabilities more efficiently.
- **Fusion & Analysis**. This module is focused in preprocessing the information obtained from the SUT, meaning

that it should be able to process the following types of responses:

- Data Response. It is the response caused by a specific input, it is the traditional response collected by fuzzers.
- Physical response. Using physical responses as a novel information source for the embedded fuzzer would allow to infer when a fault is triggered. The physical responses that can be obtained from a SUT are varied (time, electromagnetic signals etc.). These variables can help detect the state of the system, and they can be measured externally without influencing the system and they can provide information regarding its behavior.
- Hardware status monitoring. Apart from the system physical response, it is also promising to find additional sources for SUT monitoring. One example of such additional source is the Joint Test Action Group (JTAG) interface. The JTAG is a standard interface for testing to debugging embedded systems, and it is available in a wide range of devices. Therefore, using the JTAG would allow to monitor the internal status of the embedded software without instrumentation nor emulation.

Moreover, if the fuzzers needs to process different types of signals and responses, it is necessary to develop a **translator** that is able to convert incoming data to a standard set of features that the fuzzer monitor can use for examination. This translator could later be reused by other similar fuzzers.

Finally, through the use of **binary sanitization**, particularly the more promising **static** variant, would allow embedded fuzzers to have yet another information source to also infer memory violations caused by malformed inputs. Further research into binary static sanitization techniques, with lower overhead, coverage for other types of software faults and signal communication with the fuzzer would vastly improve the fuzzer's ability to detect errors in the running binary.

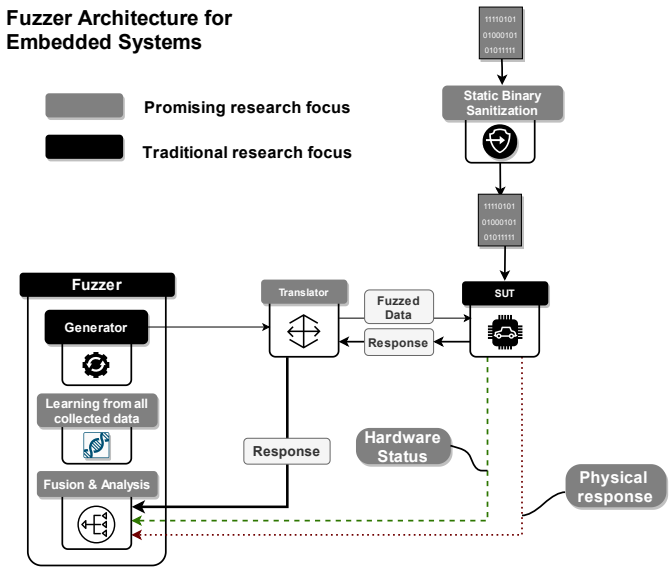

Fig. 12. Enhanced fuzzing architecture for embedded systems

C. IoT and embedded fuzzing algorithms

The embedded system particularities mentioned in Section II, mean that it is necessary to maximize fuzzer performance against the constraints of embedded systems, as well as being able to fuzz the embedded system through additional communication channels.

1) Constraint-aware fuzzers: Embedded fuzzers must be aware that the waiting time between sending a message and receiving its response is generally longer in embedded systems than in general-purpose ones. Additionally, it is possible for different tests to have different waiting times. As a consequence, responses from the SUT can overlap or come in different order. In this case, discerning what is the response from a specific input is not a trivial task. Minimizing response confusion while maximizing testing rate is an open research area that would maximize fuzzing performance.

Moreover, even with testing rate maximization, in some cases it might not be viable to fuzz the embedded system for an indeterminate time length to find as much vulnerabilities as possible. Embedded system testing time normally comes defined by the shipping deadline. Therefore, embedded fuzzing algorithms must thrive to find as much vulnerabilities as possible in this testing stage, before the product is deployed. As a consequence, it is necessary to continue improving fuzzing algorithms to optimize seed generation and mutation strategies. That includes using and further developing techniques such as static analysis, grammar representation or genetic algorithms. This research direction goes hand in hand with improvements in general fuzzing research.

2) Wireless fuzzing: While fuzzing in general-purpose systems is generally performed locally or over fast wired connections when remote, an holistic stance of fuzzing embedded IoT devices requires to consider additional communication channels. While Frankenstein [107] is a example of Bluetooth fuzzing, it is necessary to extend fuzzing to wireless protocols such as Bluetooth Low Energy [152], LoRaWAN [153], ZigBee [154] and Z-Wave. These protocols are widespread in different IoT use cases and are known to have security issues [155]. Nevertheless, the scope can be extended to virtually any IoT wireless communication protocol by using Software Defined Radio [156], opening a new research field on its own. Developing novel fuzzing approaches through wireless channels would open the possibility of fuzzing devices that lack other types of external communication, as well as testing the robustness of software drivers that enable this type of communication in devices that support it.

VII. Conclusion

With the increase of the attacks towards IoT devices it is necessary to find and solve the vulnerabilities before bringing the product to the market. In this way, new testing techniques have been sought to find new vulnerabilities more effectively. Fuzzing is one of such options that could be applied for this end. However, the particularities of IoT devices, embedded systems with limited computing power, presents a unique challenge to general-purpose fuzzers, designed to work against traditional IT targets.

The variety on embedded systems causes difficulties to fuzz some of them, particularly the most resources constrained ones. Those systems does not have an OS to trigger a signal when an error occurs, usually they do not detect the error and they continue with the execution. In addition, they do not usually have any additional hardware protection element, this will complicate the monitoring of the SUT during fuzzing. So they will need other data to detect any change in the state of the SUT, as the physical response to detect them.

In this review paper, we have described the particularities of the embedded systems, listed different fuzzing techniques and tools and classified them according to different criteria. We have also covered the constraints fuzzers should meet in order to be used against embedded systems and reviewed existing proposals according to these constraints. Finally, we have listed a set of qualities that any embedded software fuzzer should meet and identified a set of future research lines in the field.

In conclusion, the major problem to fuzz an embedded system is the monitoring. The lack of hardware protection elements or an operating system difficult the early detection of errors, so when testing those systems is necessary to consider it. However, most of the fuzzing techniques that are used nowadays do not consider it, sin they are developed for general-purpose systems.

ACKNOWLEDGMENT

We thank the anonymous reviewers whose many comments and suggestions helped improve and clarify this manuscript.

This work has been partially supported by the Department of Economic Development and Infrastructures of the Basque Government through the project "SENDAI" under Grant No.: KK-2019/00072.

Maialen Eceiza is supported through the Bikaintek program (grant no. 20-AF-W2-2019-00006) from the Department of Economic Development, Sustainability and Environment of the Basque Government.

Mikel Iturbe is part of the group of Intelligent Systems for Industrial Systems, financed by the Department of Education, Linguistic Policy and Culture of the Basque Government.

REFERENCES

[1] A. H. Ngu, M. Gutierrez, V. Metsis, and Q. Z. Sheng, "IoT Middleware: A Survey on Issues and Enabling Technologies," vol. 4, no. 1, pp. 1–20, 2017.

[2] Statista. (2019, November) IoT: number of connected devices worldwide 2012-2025. [Online]. Available: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

[3] F. Javed, M. K. Afzal, M. Sharif, and B. Kim, "Internet of things (iot) operating systems support, networking technologies, applications, and challenges: A comparative review," *IEEE Communications Surveys Tutorials*, vol. 20, no. 3, pp. 2062–2100, 2018.

[4] P. Pingale, K. Amrutkar, and S. Kulkarni, "Design aspects for upgrading firmware of a resource constrained device in the field," in *2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, 2016, pp. 903–907.

[5] Kaspersky. (2019, October) IoT under fire: Kaspersky detects more than 100 million attacks on smart devices in h1 2019.

[6] S. Babar, A. Stango, N. Prasad, J. Sen, and R. Prasad, "Proposed embedded security framework for Internet of Things (IoT)," *2011 2nd International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace and Electronic Systems Technology, Wireless VITAE 2011*, no. February, 2011.

[7] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "Ddos in the iot: Mirai and other botnets," *Computer*, vol. 50, no. 7, pp. 80–84, 2017.

[8] J. L. Beavers, M. Faulks, and J. Marchang, "Hacking nhs pacemakers: A feasibility study," pp. 206–212, 2019.

[9] A. Greenberg, "Hackers remotely kill a jeep on the highway—with me in it," *Wired*, vol. 7, p. 21, 2015.

[10] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani, "Demystifying iot security: an exhaustive survey on iot vulnerabilities and a first empirical look on internet-scale iot exploitations," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 3, pp. 2702–2733, 2019.

[11] V. Manes, H. Han, C. Han, S. Cha, M. Egele, E. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," November 2018.

[12] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," *IEEE International Conference on Software Testing, Verification and Validation*, pp. 427–430, March 2011.

[13] 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, pp. 32–44, 1990.

[14] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, January 2007.

[15] B. Z. Jun Li and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, December 2018, https://doi.org/10.1186/s42400-018-0002-y.

[16] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. PP, pp. 1–20, June 2018.

[17] R. McNally, K. Yiu, D. Grove, and D. Gerhardy, "Fuzzing: The State of the Art," *Defence Science and Technology Organisation Edinburgh (Australia)*, p. 43, 2012. [Online]. Available: https://fuzzinginfo.files.wordpress.com/2012/05/dsto-tn-1043-pr.pdf

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

[19] "IEC 62443-4-2: Requirements for Embedded Device Security Assurance (EDSA) Certification," Automation Standards Compliance Institute, Research Triangle Park, NC, Standard, 2011.

[20] "IEC 62443-4-1: Security for industrial automation and control system-Part 4-1: Secure product development lifecycle requirements," International Electrotechnical Commision," Standard, 2018.

[21] "IEC 62443-4-2: Security for industrial automation and control system-Part 4-2: Technical security requirements for IACS components," International Electrotechnical Commision," Standard, 2019.

[22] P. Amini, *Fuzzing Frameworks 21*, 2002.

[23] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang, "Fuzz testing in practice: Obstacles and solutions," *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings*, vol. 2018-March, pp. 562–566, 2018.

[24] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, vol. 75, February 2018.

[25] C. Mulliner and C. Miller, "Fuzzing the phone in your phone," 2009. [Online]. Available: https://www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf

[26] F. van den Broek, B. Hond, and A. Cedillo Torres, "Security testing of gsm implementations," in *Engineering Secure Software and Systems*, J. Jürjens, F. Piessens, and N. Bielova, Eds. Cham: Springer International Publishing, 2014, pp. 179–195.

[27] N. Kamel and J.-L. Lanet, "Analysis of http protocol implementation in smart card embedded web server," *International Journal of Information and Network Security (IJINS)*, vol. 2, August 2013.

[28] V. Alimi, S. Vernois, and C. Rosenberger, "Analysis of embedded applications by evolutionary fuzzing," July 2014.

[29] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao, "A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1125–1142, 2017.

[30] P. C. Van Oorschot and S. W. Smith, "The internet of things: Security challenges," *IEEE Security & Privacy*, vol. 17, no. 5, pp. 7–9, 2019.

[31] R. Minerva, A. Biru, and D. Rotondi, "Towards a Definition of the Internet of Things (IoT) ," https://iot.ieee.org/definition.html, 2015, [Online; accessed 11-November-2020].

[32] D. Papp, Z. Ma, and L. Buttyan, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy," *2015 13th Annual Conference on Privacy, Security and Trust, PST 2015*, pp. 145–152, 2015.

[33] ST, "User manual STM32F429ZI," no. September, 2017-09.

[34] R. Pi, "Raspberry Pi 4 Model B," no. June, 2019-06.

[35] T. Macho, "Base principles of embedded systems design," *IFAC Programmable Devices and Embedded Systems*, vol. 39, no. 21, pp. 232–235, 2006.
[36] J.-P. Vasseur and A. Dunkels, *Interconecting Smart Objects With IP: the next internet*, 2010.
[37] S. K. V., *Introduction To Embedded Systems 1E*. McGraw-Hill Education (India) Pvt Limited, 2009.
[38] C. Main. (2010, April) Real-time and general-purpose operating systems unite via virtualization. [Online]. Available: https://www.embedded-computing.com/embedded-computing-design/real-time-and-general-purpose-operating-systems-unite-via-virtualization
[39] BusyBox. [Online]. Available: https://www.busybox.net/about.html
[40] S. Parameswaran and T. Wolf, "Embedded systems security—an overview," *Design Autom. for Emb. Sys.*, vol. 12, pp. 173–183, September 2008.
[41] FreeRTOS - market leading RTOS (Real Time Operating System) for embedded systems with internet of things extensions. [Online]. Available: www.freertos.org
[42] W. R. Systems. VxWorks. [Online]. Available: https://www.windriver.com/products/vxworks/
[43] E. Decker. TinyOS. [Online]. Available: https://github.com/tinyos/tinyos-main
[44] T. Xu, J. B. Wendt, and M. Potkonjak, "Security of IoT systems: Design challenges and opportunities," *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, vol. 2015-January, no. January, pp. 417–423, 2015.
[45] J. Kindervag, "Build security into your network's dna: The zero trust network architecture," *Forrester Research Inc*, pp. 1–26, 2010.
[46] M. Salehi, D. Hughes, and B. Crispo, "μSBS: Static binary sanitization of bare-metal embedded devices for fault observability," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, October 2020, pp. 381–395. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/salehi
[47] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu, "V-fuzz: Vulnerability-oriented evolutionary fuzzing," January 2019.
[48] J. Zhao, Y. Wen, and G. Zhao, "H-fuzzing: A new heuristic method for fuzzing data generation," pp. 32–43, January 2011.
[49] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire : Data-Driven Seed Generation for Fuzzing," *IEEE Symposium on Security and Privacy*, 2017.
[50] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 206–215, 2008.
[51] A. D. Householder and J. M. Foote, "Probability-Based Parameter Selection for Black-Box Fuzz Testing," *Software Engineering Institute*, no. August, pp. 1–30, 2012. [Online]. Available: http://repository.cmu.edu/sei/691
[52] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE : Whitebox Fuzzing for Security Testing," *ACM Queue*, vol. 10, January 2012.
[53] H. Istvan, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowser: a guided fuzzer to find buffer overflow vulnerabilities," *Journal of Applied Research in Intellectual Disabilities*, vol. 31, no. 5, pp. 820–832, 2018.
[54] Google, "AFL," February 2020. [Online]. Available: https://github.com/google/AFL
[55] G. Saavedra, K. Rodhouse, D. Dunlavy, and P. Kegelmeyer, "A review of machine learning applications in fuzzing," July 2019.
[56] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," *Proceedings - 2012 4th International Conference on Multimedia and Security, MINES 2012*, pp. 152–156, November 2012.
[57] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2018-May, pp. 711–725, 2018.
[58] P. Amini, "OpenRCE/sulley," original-date: 2012-04-03T15:28:57Z. [Online]. Available: https://github.com/OpenRCE/sulley
[59] Peach community edition. [Online]. Available: https://www.peach.tech/resources/peachcommunity/
[60] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for os kernels," August 2017.
[61] Y. Chen, C. Poskitt, J. Sun, S. Adepu, and F. Zhang, "Learning-guided network fuzzing for testing cyber-physical system defences," September 2019.
[62] S. Rawat and L. Mounier, "Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: Few preliminary results," March 2011.
[63] P. Godefroid, H. Peleg, and R. Singh, "Learn & fuzz: Machine learning for input fuzzing," pp. 50–59, October 2017.
[64] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, and X. Wang, "IoTFUZZER : Discovering Memory Corruptions in IoT Through App-based Fuzzing," *Network and Distributed Systems Security (NDSS) Symposium 2018*, no. February 2018, 2020.
[65] D. Molnar, X. C. Li, and D. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," *Proceedings of the 18th conference on USENIX security symposium*, pp. 1–24, 2009. [Online]. Available: https://www.usenix.org/legacy/events/sec09/tech/full_papers/molnar.pdf
[66] M. Böhme, T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," *ACM SIGSAC Conference*, pp. 2329–2344, October 2017.
[67] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE : Whitebox Fuzzing for Security Testing," 2012.
[68] S. Sargsyan, S. Kurmangaleev, J. Hakobyan, M. Mehrabyan, S. Asryan, and H. Movsisyan, "Directed fuzzing based on program dynamic instrumentation," pp. 30–33, March 2019.
[69] H. Liang, Y. Zhang, Y. Yu, Z. Xie, and L. Jiang, "Sequence coverage directed greybox fuzzing," pp. 249–259, May 2019.
[70] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," *Journal of Applied Research in Intellectual Disabilities*, vol. 31, no. 5, pp. 820–832, 2018.
[71] NIST. NVD - vulnerabilities. [Online]. Available: https://nvd.nist.gov/vuln
[72] S. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," *IEEE Symposium on Security and Privacy*, vol. 2015, pp. 725–741, July 2015.
[73] Y. Wang, P. Jia, L. Liu, and J. Liu, "A systematic review of fuzzing based on machine learning techniques," August 2019.
[74] S. Ognawala, A. Petrovska, and K. Beckers, "An exploratory survey of hybrid testing techniques involving symbolic execution and fuzzing," December 2017.
[75] L. Zhang and V. Thing, "A hybrid symbolic execution assisted fuzzing method," *TENCON 2017 - 2017 IEEE Region 10 Conference*, pp. 822–825, November 2017.
[76] J. Fell, "A review of fuzzing tools and methods," *PenTest Magazine*, March 2017.
[77] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*, vol. 8, pp. 209–224, January 2008.
[78] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," *NDSS Symposium*, no. March, 2017.
[79] G. Klees, A. Ruef, and M. Hicks, "Evaluating Fuzz Testing," *CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.
[80] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," *IEEE Symposium on Security and Privacy (SP)*, pp. 697–710, May 2018.
[81] X. Xie, S. See, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, and J. Yin, "Deephunter: a coverage-guided fuzz testing framework for deep neural networks," *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 146–157, July 2019.
[82] 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," *NDSS*, vol. 16, no. 2016, pp. 1–16, 2016.
[83] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 511–522, 2013.
[84] M. Neugschwandtner, P. M. Comparetti, and H. Bos, "The BORG : Nanoprobing Binaries for Buffer Overreads," 2015.
[85] P. Larranaga, C. Kuijpers, R. Murga, I. Inza, and S. Dizdarevic, "Genetic algorithms for the travelling salesman problem: A review of representations and operators," *Artificial Intelligence Review*, vol. 13, pp. 129–170, January 1999.
[86] S. Mirjalili, *Evolutionary Algorithms and Neural Networks. Genetic Algorithm*, January 2019.
[87] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner, "Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach," November 2017.

[88] M. Böhme, "Aflfast." [Online]. Available: https://github.com/mboehme/aflfast
[89] M. Böhme, V.-t. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," *IEEE Transactions on Software Engineering*, vol. XX, no. X, 2017.
[90] Y. Li, B. Chen, M. Chandramohan, S. W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. Part F130154, pp. 627–637, 2017.
[91] "Angora fuzzer." [Online]. Available: https://github.com/AngoraFuzzer/Angora
[92] Google. Honggfuzz. [Online]. Available: https://github.com/google/honggfuzz
[93] H. Aki. Radamsa. [Online]. Available: https://gitlab.com/akihe/radamsa
[94] R. Zurawski, *Embedded Systems Handbook: Embedded systems design and verification*. CRC press, 2018.
[95] T. Adegbija and A. Gordon-Ross, "TaPT: Temperature-aware dynamic cache optimization for embedded systems," *Computers*, vol. 7, no. 1, pp. 1–19, 2018.
[96] A. Costin, A. Zarras, and A. Francillon, "Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces," *arXiv e-prints*, p. arXiv:1511.03609, November 2015.
[97] D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards automated dynamic analysis for linux-based embedded firmware," January 2016.
[98] M. Kammerstetter, D. Burian, and W. Kastner, "Embedded security testing with peripheral device caching and runtime program state approximation," in *SECURWARE 2016*, 2016.
[99] K. Hazelwood and A. Klauser, "A dynamic binary instrumentation engine for the arm architecture," *CASES 2006: International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 261–270, January 2006.
[100] S. Nagy and M. Hicks, "Full-speed Fuzzing : Reducing Fuzzing Overhead through Coverage-guided Tracing," *IEEE Symposium on Security and Privacy (SP)*, no. 1, 2019.
[101] P. Chen, J. Liu, and H. Chen, "Matryoshka: fuzzing deeply nested branches," *CCS '19: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, May 2019.
[102] Y. D. Lin, F. Z. Liao, S. K. Huang, and Y. C. Lai, "Browser fuzzing by scheduled mutation and generation of document object models," *Proceedings - International Carnahan Conference on Security Technology*, vol. 2015-Janua, pp. 1–6, 2016.
[103] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," *Proceedings - International Conference on Software Engineering*, pp. 474–484, 2009.
[104] Z. Wang, Y. Zhang, Z. Tian, Q. Ruan, T. Liu, H. Wang, Z. Liu, J. Lin, B. Fang, and W. Shi, "Automated vulnerability discovery and exploitation in the internet of things," *Sensors (Switzerland)*, vol. 19, no. 15, pp. 1–23, 2019.
[105] I. Pustogarov, T. Ristenpart, and V. Shmatikov, "Using program analysis to synthesize sensor spoofing attacks," *ASIA CCS 2017 - Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security*, pp. 757–770, 2017.
[106] D. Davidson, B. Moench, S. Jha, and T. Ristenpart, "FIE on Firmware: Finding Vulnerabilities in Embedded Systems using Symbolic Execution," *Proceedings of the 22nd USENIX Security Symposium*, pp. 463–478, 2013. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson
[107] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets," in *29th USENIX Security Symposium*, 2020, pp. 19–36.
[108] Google. libfuzzer and afl. [Online]. Available: /clusterfuzz/setting-up-fuzzing/libfuzzer-and-afl/
[109] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, Z. Su, and X. Jiao, "EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers," *SEC'19: Proceedings of the 28th USENIX Conference on Security Symposium*, 2019.
[110] V.-t. Pham and M. Böhme, "Model-Based Whitebox Fuzzing for Program Binaries," *ASE 2016: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 552–562, 2016.
[111] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, "PAFL: Extend fuzzing optimizations of single mode to industrial parallel mode," *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 809–814, 2018.
[112] PeachTech. Peach Fuzzer Platform Whitepaper. [Online]. Available: https://www.peach.tech/products/peach-fuzzer/peach-pits/
[113] G. Grieco, M. Ceresa, A. Mista, and P. Buiras, "QuickFuzz testing for fun and profit," *Journal of Systems and Software*, vol. 134, pp. 340–354, 2017.
[114] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," *NDSS Symposium*, 2019.
[115] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pp. 265–278, 2011.
[116] P. Godefroid, M. Y. Levin, and Mol, "Automated whitebox fuzz testing," *Network and Distributed System Security Symposium*, no. July, pp. 1–1, 2008.
[117] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, "SLF: Fuzzing without Valid Seed Inputs," *Proceedings - International Conference on Software Engineering*, vol. 2019-May, pp. 712–723, 2019.
[118] A. Warren, "Using Sulley to Protocol Fuzz for Linux Software," 2016.
[119] "google/syzkaller," original-date: 2015-10-12T06:05:05Z. [Online]. Available: https://github.com/google/syzkaller
[120] D. Li and H. Chen, "FastSyzkaller: Improving Fuzz Efficiency for Linux Kernel Fuzzing," *Journal of Physics: Conference Series*, vol. 1176, no. 2, 2019.
[121] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," *Proceedings - IEEE Symposium on Security and Privacy*, pp. 497–512, 2010.
[122] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano, "T-Fuzz: Model-based fuzzing for robustness testing of telecommunication protocols," *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014*, vol. 3, pp. 323–332, 2014.
[123] 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 *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 110–121.
[124] TCPDUMP/LIBPCAP public repository. [Online]. Available: https://www.tcpdump.org/
[125] M. Adams. Jasper. [Online]. Available: https://www.ece.uvic.ca/~frodo/jasper/
[126] F. G. G. Meade, "Juliet Test Suite v1 . 2 for C / C ++ User Guide," no. December, 2012. [Online]. Available: https://samate.nist.gov/SRD/resources/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf
[127] ARM. SSL library mbed TLS-PolarSSL. [Online]. Available: https://tls.mbed.org/
[128] J. Somorovsky, "Systematic Fuzzing and Testing of TLS Libraries," 2016.
[129] S. Pipping. libexpat/libexpat. [Online]. Available: https://github.com/libexpat/libexpat
[130] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-Scale Automated Vulnerability Addition," *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, pp. 110–121, 2016.
[131] M. Böhme, T. Pham, and M.-D. Nguyen, "Aflgo." [Online]. Available: https://github.com/aflgo/aflgo
[132] Y. Ghassemi, "Driller," original-date: 2016-08-20. [Online]. Available: https://github.com/shellphish/driller
[133] M. Ceresa, "QuickFuzz," original-date: 2015-06-27. [Online]. Available: https://github.com/CIFASIS/QuickFuzz
[134] D. Molnar, "SmartFuzz," original-date: 2009-06-29. [Online]. Available: https://github.com/dmolnar/SmartFuzz
[135] VUSec, "Vusec/vuzzer." [Online]. Available: https://github.com/vusec/vuzzer
[136] S. Hajnoczi and A. Oleinik. Fuzzing QEMU device emulation - QEMU.
[137] A. Beckus, "beckus/qemu_stm32," original-date: 2012-10-10. [Online]. Available: https://github.com/beckus/qemu_stm32
[138] AFLplus. UnicornAFL, url = https://github.com/AFLplusplus/unicornafl, urldate = 2020-11-17.
[139] E. Gustafson. Hal-fuzz. [Online]. Available: https://github.com/ucsb-seclab/hal-fuzz
[140] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling near-real-time dynamic analyses of embedded systems," *9th USENIX Workshop on Offensive Technologies, WOOT 2015*, 2015.
[141] google. Sanitizers. [Online]. Available: https://github.com/google/sanitizers
[142] Google. Address sanitizers. [Online]. Available: https://github.com/google/sanitizers/wiki/AddressSanitizer

[143] google. Thread sanitizer. [Online]. Available: https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual
[144] Google. Memory sanitizer. [Online]. Available: https://github.com/google/sanitizers/wiki/MemorySanitizer
[145] Undefined Behavior Sanitizer (UBS). [Online]. Available: https://github.com/llvm-mirror/clang
[146] ClangTeam. DataFlowSanitizer design document-clang 12 documentation. [Online]. Available: https://clang.llvm.org/docs/DataFlowSanitizerDesign.html
[147] C. Team. LeakSanitizer-clang 12 documentation. [Online]. Available: https://clang.llvm.org/docs/LeakSanitizer.html
[148] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," p. 12. [Online]. Available: http://web.stanford.edu/class/cs343/resources/valgrind.pdf
[149] O. Levi. Pin-a dynamic binary instrumentation tool. [Online]. Available: https://www.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html
[150] K. Huey, "DynamoRIO/dynamorio." [Online]. Available: https://github.com/DynamoRIO/dynamorio
[151] K. Zhou, "Dyninst." [Online]. Available: https://github.com/dyninst/dyninst
[152] 1. introduction - getting started with bluetooth low energy [book]. [Online]. Available: https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch01.html
[153] About LoRaWAN® | LoRa alliance®. [Online]. Available: https://lora-alliance.org/about-lorawan
[154] Zigbee. [Online]. Available: https://zigbeealliance.org/solution/zigbee/
[155] R. Krejčí, O. Hujňák, and M. Švepeš, "Security survey of the iot wireless protocols," in *2017 25th Telecommunication Forum (TELFOR)*. IEEE, 2017, pp. 1–4.
[156] D. Sinha, A. K. Verma, and S. Kumar, "Software defined radio: Operation, challenges and possible solutions," *Proceedings of the 10th International Conference on Intelligent Systems and Control, ISCO 2016*, no. November, 2016.

Maialen Eceiza received the degree of electronic and automatic industrial engineering in 2017 and the master of embedded systems in 2019 from the University of Basque Country. She is currectly doing the PhD at Ikerlan Technology Research Center in the Cybersecurity in Embedded System team.

Jose Luis Flores is a researcher at Ikerlan Technology Research Center and he is currently part of the Cybersecurity in Embedded System team. He holds a MSc in Robotics and Advanced Control from the University of the Basque Country. His main interest is related to Artificial Intelligence and Cybersecurity. As such, the main lines he works on in each organization are Embedded System security at Ikerlan, Machine Learning and Optimization at the university.

Mikel Iturbe is a lecturer and researcher at Mondragon Unibertsitatea and he is currently part of the Data Analysis and Cybersecurity research group. He holds a PhD from Mondragon Unibertsitatea, where he worked on data-driven intrusion detection in industrial networks and a MSc in ICT Security from the Open University of Catalonia. His main research interest is related to cybersecurity, primarily in the industrial sector. As such, the main lines he works on are Industrial Control System security, Embedded Security and Software Security. He also works in exploring novel data-driven applications for cybersecurity.