UNIVERSITÄT
DES
SAARLANDES

Debugger-driven Embedded Fuzzing

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

submitted by
Max Camillo Eisele

Saarbrücken, 2024

Abstract

Embedded Systems – hidden computers – are all around. Programed by humans or, eventually, artificial intelligences, embedded systems run software to enrich, entertain, and evaluate our lives on all imaginable grounds. Safety-critical systems, such as vehicles or industrial production plants, feature numerous bespoke embedded computers whose software monitors, measures, and manipulates things in their environment. Since programmers and artificial intelligences make mistakes, software contains errors. Software testing is therefore indispensable for finding programming flaws, ideally before deployment.

Automated software testing methods can automatically generate test data for programs to assist test engineers with the task of crafting test cases. However, available techniques are primarily established for testing applications on personal computers and servers. Deploying automated software testing techniques on embedded systems is subject to additional challenges, mainly arising from the low number of shared communalities in terms of interfaces, peripherals, as well as hardware and software architectures. This thesis examines obstacles to fuzzing embedded systems and defects of state-of-the-art approaches. Furthermore, it introduces two new methods for fuzz testing embedded systems, overcoming the distilled defects. Also, both methods leverage only generic and widespread features for analyzing embedded programs during runtime and thus are applicable on a variety of devices in practice.

Zusammenfassung

Eingebettete Systeme – versteckte elektronische Rechner – sind allgegenwärtig. Programmiert von Menschen, oder zukünftig auch von künstlichen Intelligenzen, führen eingebettete Systeme Programme aus, um menschliches Leben zu bereichern, zu belustigen und zu bewerten. Sicherheitskritische Systeme, wie Fahrzeuge oder Industrieanlagen, enthalten zahlreiche maßgeschneiderte eingebettete Rechner, deren Programme Dinge in ihrer Umgebung mustern, messen und manipulieren. Da Programmierer und künstliche Intelligenzen Fehler machen, können Programme fehlerhaft sein. Gründliches Testen von Programmen ist deshalb unerlässlich, idealerweise bevor diese ausgeliefert werden.

Automatisierte Programmtestverfahren können automatisch Testdaten für Programme erzeugen, um Testingenieure beim Erstellen von Testfällen zu unterstützen. Die verfügbaren Techniken sind jedoch in erster Linie für das Testen von Anwendungen auf traditionellen Heim- und Dienstgeberrechnern gedacht. Diese auf eingebetteten Rechnern anzuwenden ist mit zusätzlichen Herausforderungen verbunden, die sich hauptsächlich aus der geringen Anzahl von Gemeinsamkeiten in Bezug auf Schnittstellen und Peripherie, sowie durch Verwendung verschiedener Architekturen ergeben. Diese Dissertation untersucht die Herausforderungen automatisierte Programmtestverfahren auf eingebetteten Rechnern auszuführen und deckt Lücken im Stand der Technik auf. Des Weiteren stellt sie zwei neue Methoden vor, welche diese Lücken schließen sollen. Beide Methoden nutzen nur weit verbreitete Funktionalitäten, um Programme auf eingebetteten Rechnern während der Laufzeit zu analysieren und sind deshalb auf einer Vielzahl von Geräten in der Praxis anwendbar.

Acknowledgements

I would like to express my deepest gratitude to my advisor, Prof. Dr. Andreas Zeller, for his unwavering trust, continuous support, and invaluable guidance throughout the past years. His mentorship has been instrumental in shaping this dissertation.

I am deeply grateful to my reviewers, Prof. Dr. Martina Maggio and Prof. Dr. Thorsten Holz, for taking on the task of evaluating my work and for the rich discussions in the colloquium.

I would like to thank my supervisor at Bosch, Dr. Christopher Huth, for his guidance and support. I am also deeply grateful to Dr. Irina Nicolae and Dr. Martin Ring for their collaboration and shared thoughts, which greatly enhanced the quality of this work.

I am very thankful to all members of the research group of Prof. Dr. Andreas Zeller for their stimulating discussions and the collaborative environment that has been fostering my work.

To my friends, the Bahlinger Jungs, especially my best friend Mario, thank you for always keeping me grounded. Your friendship has been a source of joy and balance in my life.

To my former fellow students, Markus and Valentin, thank you for the endless technical discussions that broadened my perspective and deepened my understanding. Your intellectual company has been invaluable.

A heartfelt thanks to the Kellner de Cádiz, the Karpathos Burschn, and the Kitekrise Gang for enriching my time with unforgettable experiences and cherished memories. Your companionship has added a wonderful dimension to my life.

The gymnastics club WTG Heckengäu has been instrumental in keeping me fit and motivated throughout this journey.

To my partner Laura, your love, patience, and encouragement have been my anchor throughout this journey. Thank you for standing by my side through thick and thin. Your unwavering support has been a cornerstone of my success.

I would like to express my deepest gratitude to my parents, Sonja and Michael, for their tireless support and unconditional trust throughout more than three decades. Your belief in me has been a constant source of motivation. I am also thankful to my brother Janik and the rest of my family for always providing me with a safe haven and a happy home.

Lastly, I would like to thank my grandfather Karl-Heinz for encouraging me to be as curious as I am today. Your wisdom and guidance have been a profound influence on my life.

Thank you all for your unwavering support and encouragement. This dissertation would not have been possible without you.

Contents

1 Introduction

Software is what makes devices appear *smart*. The software joins the circuits on a processor in such a way that inputs get interpreted and logically processed into outputs. From a theoretical perspective, it is undecidable whether non-trivial software programs behave *well* for any input. Nevertheless, developers want to ensure proper function of software before shipping. Best practice since decades has been developers creating tests that execute the program or parts of it with different inputs and verify the outputs for these [PY08]. The input space of programs, however, grows exponentially with the amount of *bits* required to represent it. Already a program input of 33 bytes[1] takes $2^{264} \approx 10^{79}$ unique bit combinations, which is enough to number each of the estimated 10^{78} protons in the universe [Edd31]. Many programs even digest arbitrarily large inputs, making it infeasible to exercise through all possible concrete inputs. Consequently, tests only cover a subset of a program's input space and manually creating test inputs for covering large program parts is laborious.

Fuzzing promises relief – at least to the manual task of creating test inputs. Fuzzing techniques generate massive amounts of randomized inputs to test software. Basically, a fuzzer tries to brute force inputs that cause the software to crash. Since purely random generated input has small chances of reaching large parts of the code, sophisticated fuzzing tools make use of additional information. Knowledge of the input structure or dynamically extracted code coverage information can guide fuzzers to generate *rich* test inputs [LZZ18]. Attractively, fuzzing is an unsupervised method. Once set up, a software developer can focus on other tasks while the fuzzer finds bugs in background.

Despite its simple underlying principles, fuzzing demonstrated its ability to detect programming errors in practice. Google alone lists tens of thousands of bugs found by fuzzing within their *OSS-Fuzz* initiative [Goo16b]. Several industry standards now recommend fuzzing as one of the testing methods to ensure robustness [Int18b; Int18a], including the recently released *ISO/SAE 21434 - Road vehicles - Cybersecurity* [Int21].

Fuzzing user space applications for general purpose operating systems on personal computers or servers is perhaps the best-established use of fuzzing in practice. However, this addresses only a fraction of electronic devices that execute software. *Embedded systems* outnumber personal computers and servers by far and are used pervasively in modern society, for instance in smart meters, pacemakers, vehicles, and factory robots, to name just a few. More precisely, general-purpose computers and servers account for less than 1% of all the microprocessors sold every year [Gar23; SI19; IC 22] and almost all deployed microprocessors in the world end up in embedded systems. While the embedded computing market grows constantly and is expected to continue this trend in the near future [Als19], code bases in industry grow steadily,

[1]An empty Microsoft Word document takes multiple thousand bytes.

too [Sac22]. Manual software testing is laborious and the growth of deployments and code yields a significant demand for effective *embedded fuzzing* methods.

This thesis is about applying fuzzing on embedded systems, which is challenging. A large share of recently published embedded fuzzing approaches relies on *virtualization* of the embedded system [Yun+22; Eis+22], benefitting from easily accessible insights to program execution and scalability. Such virtualization, however, requires a trade-off between speed and fidelity [Fas+21; Wri+21]. Worse even, it requires not only virtualization of the microprocessor itself, but also of *all other hardware components on the board* as well as virtualizing the way they communicate with each other. Given the enormous *diversity* of available hardware peripherals [Hea02], this requires considerable setup and configuration costs, if possible at all. Alternative hardware-based embedded fuzzing methods use exotic or hardware-specific features, such as hardware tracers, lacking broad applicability.

The following restrictions make many of the currently existing fuzzing techniques impractical to non-applicable on embedded systems:

- The executable code may come in read-only or very constraint memory, and thus cannot be modified to obtain insights like coverage information during runtime.
- There may be no storage or channel available to feed back runtime information.
- Embedded firmware is tightly coupled with the specific hardware, hampering virtualization which could provide desired runtime information.
- Code and components may come from third parties, with lack of source code and limited documentation available.
- Input formats specifications might be unknown.
- There is a staggering variety of languages, processors, peripherals, and operating systems preventing a one-fits-all solution.

1.1 Motivation

In 2019, the security researcher Anna Prosvetova found that via exploiting critical vulnerabilities in the Wi-Fi driver of a *smart* pet feeding machine an attacker can modify the firmware of the enclosed microcontroller[2], giving an attacker full control over the feeding schedule. While an average house cat will probably make its way to find alternative food sources until their masters recognize the machines' malfunction, such vulnerabilities can cause severe damage in other embedded system appliances.

As every product, the ones including embedded systems are subject to rules of the free market economy. Testing takes time and therefore is expensive. This leads to a tradeoff between thorough software development and the acceptance of errors in the products sold. According to Rollbar's State of Software Code Report 2021 [Rol21], 88% of software errors are caught after deployment by users and 86% of all developers wish for better tools to detect and fix

[2]https://www.zdnet.com/article/security-researcher-gets-access-to-all-furrytail-pet-feeders-around-the-world/

errors. Consequently, to be considered, software testing methods for embedded systems must be efficient and be applicable with low efforts. The motivation of this thesis is to fulfil this demand by providing sophisticated fuzz testing methods for embedded systems.

1.2 Contribution

Decades of research and development on virtualization of hardware did not lead to accurate emulators for arbitrary devices. Powerful hardware tracing methods, which enable fuzzing on the actual hardware, are costly and rarely available. Crafting input specifications in a fuzzer usable format is time-consuming. This thesis proposes embedded fuzzing solutions that do not demand such hardly available features or specifications. Boiling down the common characteristics of embedded devices, it turns out that breakpoints can interrupt the execution on reaching program addresses and watchpoints can do similar on memory accesses. Both are deployable via generic interfaces. Strategic use of breakpoints and watchpoints helps programmers to analyze a program during runtime. Hardware breakpoints and watchpoints correspond to actual registers in the debug logic of the microcontroller. Consequently, the number of simultaneously deployable breakpoints and watchpoints is limited. This thesis presents new methods that require only access to limited numbers of breakpoints and watchpoints to enable sophisticated fuzzing of embedded systems on the hardware itself.

Specifically, this thesis' contributions are:

- A thorough *review* and discussion of state-of-the-art solutions for embedded fuzzing.
- GDBFuzz: Using the limited number of available hardware breakpoints in microcontrollers for coverage-guided fuzzing.
- GDBMiner: Using limited amounts of hardware watchpoints for data-flow analysis to learn context-free grammars that are usable for generation-based fuzzing.

1.3 Publications

The following publications arose during working on this dissertation, and they build the foundation of this work and are reflected to a large extent in it.

- **Max Eisele**, Marcello Maugeri, Rachna Shriwas, Christopher Huth, and Giampaolo Bella. „Embedded fuzzing: a review of challenges, tools, and solutions“. In: *Springer Cybersecurity* (2022).

  In this literature review, we investigate and taxonomize the current state of the art of embedded fuzzing approaches. Particularly, we present and summarize the top 42 approaches we feel are most relevant and discuss their advantages and disadvantages. We hypothesize that there is no *golden tool* for embedded fuzzing, and we reveal the demand for more practically and broadly applicable solutions. Chapter 3 reflects the contents of this paper.

- **Max Eisele**. „Debugger-driven Embedded Fuzzing“. In: *2022 IEEE International Conference on Software Testing, Verification and Validation (ICST).* **(Best Submission Award)**. 2022.

  With this paper on the *Doctoral Symposium* of the 2022 International Conference on Software Testing, Verification and Validation, I first present the concept of approaching common debug interfaces of microcontrollers with the *GNU Debugger (GDB)* tool to build generic embedded fuzzing solutions. I list abstract debug features, such as hardware breakpoints, hardware watchpoints, and single-stepping, and outline that these can enable partial coverage-guidance for fuzzing, learning input specifications from sample inputs, and concolic execution. The ideas of this paper led to the next two publications.

- **Max Eisele**, Daniel Ebert, Christopher Huth, and Andreas Zeller. „Fuzzing Embedded Systems Using Debug Interfaces“. In: *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023).* 2023.

  In this paper, we present GDBFuzz, which enables coverage-guided fuzzing for any system using hardware breakpoints. GDBFuzz distributes the limited amount of hardware breakpoints randomly on the basic blocks of the target program to obtain partial code coverage information. Furthermore, it boosts the limited execution insights by using dominator relations derived from the control flow graphs of the target programs. We show in the evaluation that GDBFuzz reaches more code in less time compared to fuzzing without feedback, is applicable to many diverse microcontroller setups, and performs where other solutions fail. Chapter 4 features the concept, implementation details, and evaluation from this paper.

- **Max Eisele**, Johannes Hägele, Christopher Huth, and Andreas Zeller. „GDBMiner: Mining Precise Input Grammars on (almost) any System“. In: *Under Submission.* 2024

  This paper presents GDBMiner, a tool for learning context-free grammars from any system offering common debug capabilities. It leverages stepwise execution and data watchpoints for analyzing at which program parts the target program consumes parts of the input. By finding common nodes in the resulting tree structures in a second step, GDBMiner produces context-free grammars, which can enable generation-based fuzzing. Chapter 5 describes the design, implementation, and evaluation of GDBMiner.

Additionally, the following secondary publications addressed further facets of automated software testing, but are not part of this thesis.

- Matthias Börsig, Sven Nitzsche, **Max Eisele**, Roland Gröll, Jürgen Becker, and Ingmar Baumgart. „Fuzzing Framework for ESP32 Microcontrollers“. In: *2020 IEEE International Workshop on Information Forensics and Security (WIFS).* IEEE. 2020, pp. 1–6.

- Maria Irina Nicolae, **Max Eisele**, and Andreas Zeller. „Revisiting Neural Program Smoothing for Fuzzing“. In: *Proceedings of the 31th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).* 2023.

1.4 Thesis Structure

To enhance the readability and comprehension, this thesis visually emphasizes crucial contents as:

Definitions	for providing consistent terminology.
Theorems	for fixing key facts or interrelations.
Takeaways	for capturing essential findings.

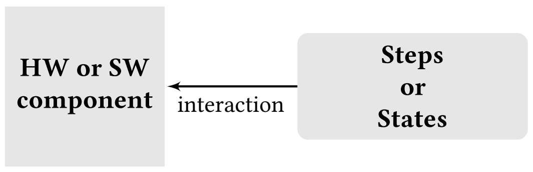


Figure 1.1: Meaning of shapes in figures throughout this dissertation.

The shapes in the figures of this thesis have consistent meanings. As shown in Figure 1.1, hardware and software components are depicted as angular boxes; algorithmic steps or states visualized as rounded boxes.

The remainder of this thesis is organized as follows. Chapter 2 explains background knowledge and concepts. Chapter 3 discusses state-of-the-art resulting from our literature review. Chapter 4 introduces the concept of GDBFuzz and Chapter 5 describes GDBMiner approach. This thesis ends with a discussion about future work and a conclusion in Chapter 6.

2 Background

This chapter describes notions, concepts, and methods that are important for this thesis. It starts with the definition of *software bugs* in Section 2.1, explains how *fuzzing* reveals bugs in Section 2.2, outlines what *embedded systems* are in Section 2.3, and how debuggers help to analyze programs in Section 2.4. The remaining Sections 2.5 to 2.7 detail the concept of program *control flow*, *dominator relations*, and *context-free grammars* in conjunction with fuzzing.

2.1 Software Bugs

The origin of the term *"bug"* regarding flaws in electronic devices lies well back in time and appears to be not fully identified. However, it is clear that Thomas Edison – inventor of the electric light bulb, the automatic telegraph and the carbon telephone transmitter – already wrote about failure inducing *bugs* in his apparatus starting from 1876 [MI13]. At that time he probably really referred to insects that caused malfunctions in his inventions, but later on he described *"fixing a bug"* as *"an expression for solving a difficulty"*. Soon after, the term was incorporated in *The Standard Electrical Dictionary* [Slo92] from 1892 and defined as *"Any fault or trouble in the connections or working of electric apparatus"*.

"Bug" is quite unspecific when it comes to software malfunctions, which is why the *ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary* [Int17] distinguishes between three terms:

Definition 1: *[Error]*

An *error* is a human action that produces an incorrect result.

In other words, *errors* are mistakes, such as programming flaws.

Definition 2: *[Fault]*

A *fault* is a manifestation of an error in software.

Consequently, programming *errors* can result in program *faults*, for instance, an incorrect output.

Definition 3: *[Failure]*

A *failure* is the termination of the ability of a system to perform a required function or its inability to perform within previously specified limits.

Faults can lead to *failures* that prevent continuing the normal execution of a program or system, for instance, because of an illegal memory access. The term *bug* suits best to the definition of *fault*, but informally programmers use it universally for errors, faults, and failures. *Debugging* or *"fixing a bug"* thus describes the process of finding root causes of a bug and the correction of corresponding programming errors.

As an example, the function `process_data` in Listing 2.1 contains a bug. It does not verify that the function parameter `length` lies within the range of the local array `stack_array`. When the first four characters of parameter `buffer` meet the constraints from line 4-7, that is they equal "bug!" and `length` is larger or equal 20, the `memcpy` function writes beyond the range of `stack_array` – it *overflows*. These *Out-of-bounds Write* bugs spearhead the *2022 CWE Top 25 Most Dangerous Software Weaknesses* [The22]. Attackers might *exploit* such bugs with cleverly crafted inputs that lead to an alternation of the execution flow and can result in arbitrary code execution. Further details about exploitation of bugs is out of scope of this thesis.

Listing 2.1: Buggy function with possible stack overflow, inspired by [GLM12].

```
1  void process_data(char* buffer, unsigned int length) {
2    char stack_array[20];
3
4    if( length > 0 && buffer[0] == 'b')
5      if( length > 1 && buffer[1] == 'u')
6        if( length > 2 && buffer[2] == 'g')
7          if( length > 3 && buffer[3] == '!')
8            memcpy(stack_array, buffer, length);
9  }
```

2.2 Fuzzing

Decades ago, Miller *et al.* [MFS90] tested Unix command line tools with random data, observed failures like crashes and hangs resulting from software errors, and called this method "fuzzing". Attractive is that the method is automated and unsupervised, meaning the testing process is solely executed by a computer, resulting in good scalability and allowing developers to focus more on creative and administrative tasks. Prerequisites for fuzzing are a well-defined way of *evaluating inputs* on a program and *bug oracles* that determine whether faults occur while the program processes inputs. Manès *et al.* [Man+19] define a bug oracle as follows:

Definition 4: *[Bug Oracle]*

A *bug oracle* is a program, perhaps as part of a fuzzer, that determines whether a given execution of the program under test violates a specific correctness policy.

Providing a distinct way of evaluating inputs on the target program is subject to a test engineer who leverages knowledge of the system or the program to connect a *fuzzer* to it. Often it requires some code for this purpose, which is called *fuzz driver* or *fuzz harness*.

Definition 5: *[Fuzz Harness]*

The *fuzz harness* is a program, mechanism, or a part of the target program that exposes a well-defined input interface for fuzzing.

In the original work, Miller generated random strings to fuzz test standard UNIX utilities via UNIX pipes and, as simple bug oracle, checked if the target programs emitted crash signals or did not finished execution within a predefined amount of time (hang) [MFS90]. Modern fuzzing setups transfer test cases via shared memory mechanisms [Swi16; Mic17; Fio+20] or compile the fuzzing logic directly into the target program [LLV15] to minimize overhead. More sophisticated fuzz harnesses might fill complex data structures with fuzzing data, invoke multiple functions of the target program, and perform error checks.

Memory fault detectors like *AddressSanitizer* [Ser+12] help to find non-crashing memory corruptions. Differential [McK98] and metamorphic [Che+18b] testing techniques can serve as bug oracles beyond failures and memory corruption, but are out of scope of this thesis.

Figure 2.1 shows how a *fuzzer* sends test inputs via a *harness* to a *target* while the *bug oracle* observes the execution correctness and collects failing inputs – *bugs*.

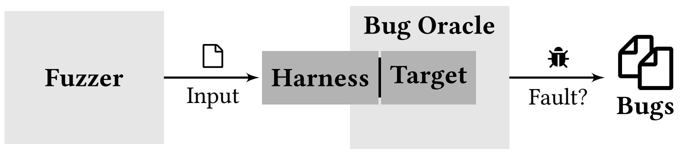


Figure 2.1: Overview of the fuzzing flow.

In practice, fuzzing is an iterative process that generates and evaluates huge amounts of inputs over time. Fuzzing campaigns over hours, days, and weeks are not uncommon [Liy+23].

Since fuzzing relies on observable faults, it can only detect bugs in code that is actually executed, which is why reaching a high code coverage is desired and reached code coverage is the most used metric in comparing fuzzers [BSM22]. With this purpose in mind, tons of different fuzzing techniques have been developed, mainly divided into *mutation-based fuzzing*, where known inputs to the program are randomly mutated, and *model-based fuzzing*, where test data is generated based on a specification of the input language [Man+19].

Additionally, literature divides fuzzers often into *blackbox*, *graybox*, and *whitebox* approaches depending on the amount of static and dynamic analysis a fuzzer performs on the target source or binary code [Man+19; LZZ18; God20]. Blackbox fuzzing keeps the target program or system unopened and uninvestigated and approaches only publicly facing interfaces. Whitebox fuzzing traditionally is to symbolically solve constraints that are statically extracted from the source code to mathematically derive inputs that trigger different code paths. Graybox fuzzing lies somewhere in between and uses lightweight analysis to guide fuzzing. However, this thesis refuses to categorize methods into graybox, and whitebox fuzzing, because they have low expressiveness and conceal important differences, especially in the realm of embedded systems. For example, code coverage feedback through source code instrumentation, as well as through binary emulation are both considered *graybox* approaches. The former requires source code and the latter not, which makes a huge difference in practice. The term *graybox*, therefore, is too vague for practical approaches and will not be used further in this thesis.

2.2.1 Mutation-based Fuzzing

Mutation-based fuzzing starts with few sample inputs for the target program, called *seeds*. The seeds are initially added to the *input corpus*, a collection of base inputs, from which mutated test inputs are generated. If no seeds are provided, the fuzzing campaign starts with an empty input corpus and generates the first input purely random. Mutation operators are, for instance, *bit flips*, *value replacements*, *random string insertions*, and *deletions*.

Mutation-based fuzzing is particularly effective when test inputs that trigger previously unseen behavior of the target are added back to the corpus, called *feedback-driven fuzzing*. Coverage-guided fuzzers consider the execution of new code paths or blocks as previously unseen behavior and thus add newly generated test cases to the corpus when they increase code coverage. Dozens of coverage-guided fuzzers exist and this flavor of fuzzing is what found tens of thousands of bugs in the *OSS-Fuzz* project [Goo16b]. Figure 2.2 shows where the feedback channel is added into the previously introduced fuzzing flow.

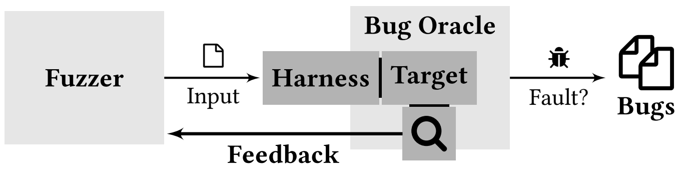


Figure 2.2: Feedback-driven fuzzing flow.

The rate of newly found code coverage during fuzzing usually decreases drastically over time. Recent work even suggests an exponential dependency between reached coverage and spent effort [BF20], which results in logarithmic appearing coverage over time plots.

Consider the code of the function `process_data` in Listing 2.1 that causes a stack buffer overflow when the first four characters of `input` match `"bug!"` and `length` has a greater value than 20. A fuzzer without feedback needs to guess the first four characters correctly from $2^{8*4} = 2^{32} = 4,294,967,296$ combinations[1] at once. A coverage-guided fuzzer can progress on each comparison step with 2^8 possible combinations individually. In summary, this lowers the number of combinations to $(1 + 2 + 3 + 4) * (4 * 2^8) = 10,240$ [BLW21], increasing the overall probability of generating an input that triggers the stack overflow during fuzzing.

Fuzz testing on general purpose operating systems often leverages *source code instrumentation,* where additionally inserted code feeds code coverage back to the fuzzer. Alternatively, *binary rewriting* or *dynamic instrumentation* tools can inject instrumentation mechanisms into compiled code or *emulators* can obtain the required code coverage feedback during runtime when no source code, and therefore no code instrumentation at compile time, is available. Control flow

[1]Assuming 8-bit characters.

graphs and corresponding measurement of code coverage is essential for this thesis and further elaborated in Section 2.5.

2.2.2 Model-based Fuzzing

Model-based fuzzers leverage knowledge of the target's input format to generate test cases from scratch that ideally cover most of the input space. There are various methods to create suitable models that allow efficient generation of test inputs [PBR16; Yan+11; SZ22; HZ19; Sor+20]. They all have in common that creating an input specification model is a manual task and that this task has to be exercised for each individual input format or protocol. A great advantage is the independence of the models, making them reusable across multiple programs e.g. a model for the PNG image format enables fuzzing for all PNG parsing programs; an SQL query model for various database management systems. Another advantage is that model-based fuzzing requires no modification of the target program or system, avoiding code instrumentation and establishing a feedback channel. Furthermore, models can define expected outputs of the target or series of inputs and outputs and thus also find non-crashing logical bugs [Per17].

A number of published methods attempt to automate the labor-intensive task of creating input models and thus try to solve the major drawback of model-based fuzzing [GMZ20; Bas+17; KLS21; SZ22]. A large portion of model-based fuzzers uses context-free grammars and since these are a substantial part of this thesis they are further explained in Section 2.7.

2.3 Embedded Systems

Literature varies on what exactly makes up an *embedded system*. One definition says that "an embedded is designed to perform a dedicated function" [Noe12], which, for instance, disqualifies personal computers or modern smartphones as such. Others define that "Embedded systems are information processing systems embedded into enclosing products" [Mar21], from which one could argue that smartphones or laptops are included. However, it is clear that an embedded systems runs software on any kind of processor or microcontroller and interacts with the outside world e.g. via sensors, actuators, or data interfaces.

Through the *embedded* character the dissemination of embedded systems is easily underestimated. Looking at the numbers, there are approximately 300 *million* annually shipped units of general purpose PCs and servers worldwide over the last 5 years [Gar23; SI19]. Over the same period, yearly shipments of microcontrollers are estimated to exceed 30 *billion* units [IC 22], meaning that about 99% of all software running devices, including smartphones, are not computers in the classical sense.

Embedded systems are highly diverse. While there might be two relevant processor architectures (x86, ARM) for personal computers and mainly three operating systems (Windows, Linux, macOS) [Sta23], there are a magnitude more in the embedded systems sector, for instance, ARM, RISC-V, MIPS, Xtensa, MSP430, and countless specialized operating systems or abstraction frameworks. This diversity impedes a detailed generic grasp of embedded systems, and leaves us with the simple minimal architectural model for embedded systems in Figure 2.3.

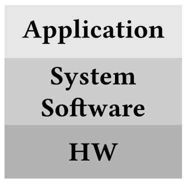


Figure 2.3: Embedded systems architecture model according to Noergaard [Noe12].

The model consists of only three layers, from which the two uppermost are software and in conjunction are referred as *firmware*. While the application layer contains program logic, the system software layer locates potential operating systems, bootloaders, drivers, and Hardware Abstraction Layer (HAL) modules. There are embedded systems without a dedicated operating system, often referred to as *bare-metal systems*. If done right, the software from the application layer might be portable over different systems and, for instance, can be cross-compiled into user applications for fuzzing. The system software, however, is tailored to the expected hardware and running it in a different environment requires major effort, such as *emulating*, *mocking*, or *stubbing* expected features. A good example therefore are drivers that access registers of the hardware directly and expect precise and timely behavior of the hardware.

2.4 Debuggers and Breakpoints

Programmers use debuggers to investigate programs at runtime, notably to understand unexpected program behavior. Debugging tools therefore can externally control the execution of a program at the programmer's pace, as well as examine memory values, such as variables or the execution stack. Specifically:

- Breakpoints on instruction addresses interrupt the execution when reached.
- Watchpoints on memory addresses interrupt the execution on memory accesses.
- Single-stepping executes a single instruction at a time only.

2.4.1 Debug Units

The hardware-side implementation of debugging capabilities depends on the processor architecture. However, commonly there are dedicated *debug units* on the processor that can control the execution on demand.

For instance, the *ARMv7-M Architecture* [ARM21] intends a Flash Patch and Breakpoint (FPB) unit and a Data Watchpoint and Trace (DWT) unit. The FPB unit can virtually patch instructions or data in the firmware by remapping it, but also features breakpoint registers that halt the execution when the program counter equals their value. The DWT unit contains comparator registers that can serve for multiple debugging tasks e.g. counting memory accesses or realize data watchpoints. ARM specifies approaching the debug units from outside via a Joint Test Action Group (JTAG) debug port [IEE13] or ARM's Serial Wire Debug (SWD) port [ARM06].

Usually a microcontroller can control its own debug units, allowing debugging of user applications from an operating system. However, particularly for embedded systems without general purpose operating systems like Linux, it is more common to attach a *debug probe* to a provided debug port and perform debug operations from the programmer's PC that also hosts the development environment and the source files of the application [Ver08].

Debug interfaces on commercial devices are ideally closed or disabled to prevent attacks. However, it has been shown several times in the past that disabled debug interfaces can be reopened, using fault injection attacks like power or clock glitching [Sko11; Kha20]. Also, the firmware from a commercial device can be transferred to an equivalent development board with accessible debug interface, and thus enable debugging.

2.4.2 GNU Debugger

A popular debugger for user programs since decades is the GNU Debugger (GDB) [S+88; LO96]. It is *versatile* and provides a *unified debug interface* for most software and systems. GDB works for Assembly, C, and C++, the top 3 languages for embedded systems [Jet22], but also supports Ada, Objective-C, Rust, Pascal, Modula-2, FORTRAN, Go, and many more compiled programming languages; it can also be applied on binaries without source code. It supports most microprocessor instruction sets, including ARM, x86, Motorola 68000, MIPS, PA-RISC, PowerPC, and RISC-V. Its GDB remote serial protocol [Gat99] is the de facto standard protocol for connecting different debug backends to various debugging frontends and is implemented by most available on-board and off-board microcontroller debug probes, for instance from Segger [Seg22], STMicroelectronics [STM19], or Lauterbach [Lau22], serving as a generic abstraction layer for debugging embedded systems.

2.4.3 Breakpoints

GDB distinguishes between *hardware breakpoints* that correspond to actual registers on the chip and *software breakpoints* that are realized by patching the binary with interrupt inducing instructions. The result is that the number of available hardware breakpoints is limited, but they can be set to any program address, regardless of the memory type the respective code is stored in. In contrast, software breakpoints are not limited in their amount, but require frequent rewriting of the program, which can be slow, can wear out eventual flash memory, or in case of read-only memory, simply be impossible.

2.4.4 Watchpoints

The situation is similar with *watchpoints* that allow observing specific variables or memory regions for read and/or write access. *Hardware watchpoints* are efficient, but come in small numbers, if at all. *Software watchpoints* are unlimited in numbers, but can require interrupting execution with every step or emulating the application, resulting in a huge performance impact. If supported by the hardware, GDB offers masking of watchpoint target addresses for covering a whole memory region with a single watchpoint [S+88].

2.5 Control Flow Graphs

Analyzing programs is usually done on the basic block level [All70].

Definition 6: *[Basic Block]*

A *basic block* is a linear sequence of program instructions having a single entry point and a single exit point.

Basic blocks can therefore contain control flow changing instructions, such as *jumps* or *conditional branches* exclusively as last instruction. Exceptions are *calls* to other functions. They interrupt the linear execution of instructions within a basic block, but because function calls are designed to return to the caller, the linear execution of the basic block is not violated. Therefore, call instructions are typically allowed in the middle of a basic block. This thesis, however, leverages an *interprocedural* view on control flow and for the sake of simplicity call instructions do terminate basic blocks in the remainder of this thesis, to allow precise reflection of call and return sites.

Definition 7: *[Control Flow Graphs]*

A *Control Flow Graph (CFG)* describes all possible control flow between the basic blocks of a program.

Local control flow graphs serve for flow analysis within functions [All70], and are constructed by ignoring control flow changes from *call* and *return* instructions. Consequently, there is one control flow graph for each function of a program. Figure 2.4 shows the local control flow graphs of the program from Listing 2.2.

Listing 2.2: Example C program.

```c
int funcA(int x) {
  return x + 5;
}

int main(int x) {
  int ret;
  if ( x & 1) {
    ret = funcA(x);
    ret--;
  } else {
    ret = funcA(-x);
    ret++;
  }
  return ret;
}
```

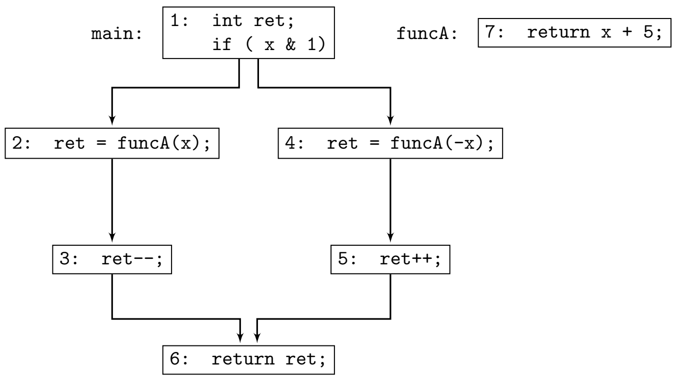


Figure 2.4: Corresponding local control flow graphs.

2.6 Dominator Relations of Control Flow Graphs

Dominator relations describe further coherence between nodes in a control flow graph. We use the notion of pre- and postdominator from Agrawal [Agr94] and assume that control flow graph G has exactly one entry point and exactly one exit point[2]:

Definition 8: *[Predomination]*

A node $u \in G$ *predominates* another node $v \in G$, denoted as $u \xrightarrow{pre} v$, if every path from the entry node to v contains u.

Definition 9: *[Postdomination]*

A node $w \in G$ *postdominates* another node $v \in G$, denoted as $w \xrightarrow{post} v$, if every path from v to the exit node contains w.

The dominator relations can be represented as (dominator) *trees* and there exist efficient algorithms to compute them from local control flow graphs [CHK01]. The postdominator tree equals the predominator tree from the reversed control flow graph [Agr94]. From any pre- and postdominator tree, we can derive the following transitive knowledge about other nodes:

Theorem 1: *[Reachability]*

If node v is reached, all parent nodes in the predominator tree have been reached before and all parent nodes in the postdominator tree will be reached afterwards.

As a result, reaching all leave nodes in the dominator tree of a program suffices to achieve 100% code coverage.

2.7 Context-Free Grammars

Context-free grammars are mathematical descriptions for context-free languages. They can serve for generating inputs that belong to the corresponding language or for verifying if an input belongs to that language. For instance, the language of correctly parenthesized expressions – for every opening parentheses there is a closing one – is a popular example of a context-free language. In contrast to regular languages, context-free languages can take care of these matching parentheses, which makes them ideal for describing mathematical expressions, but also the syntax of programming languages. Generating strings from a context-free grammar is easy, and deciding whether a string is producible by a given context-free

[2]For functions with multiple returns, the returning blocks are connected to a new virtual return block leading to a single exit node in the control flow graph.

grammar is efficient [Ear70]. Limits of context-free languages are context-sensitive properties, for instance, matching arbitrary named tags as they are used in the XML format, or checksums over the content.

Formally, a context-free grammar is a four-tuple $G = (V, T, P, S)$, with variable (or non-terminal) symbols V, terminal symbols T, the set of productions P, and the start symbol S [HMU01]. A production rule $p \in P$ is a relation (v, x) with variable $v \in V$ mapped to a string of variables and terminals $x \in (V \cup T)^*$. As an example, this grammar G represents correctly parenthesized expressions:

$$\langle S \rangle \quad ::= \quad \langle S \rangle\langle S \rangle \mid (\langle S \rangle) \mid \epsilon$$

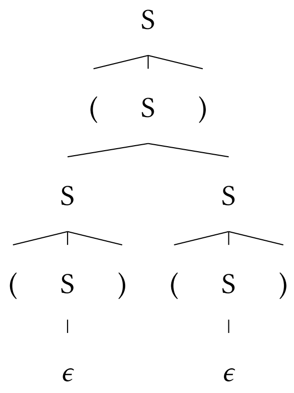


Figure 2.5: Grammar parse tree on *'(()())'*.

Applying G on the string *'(()())'* yields the parse tree in Figure 2.5.

2.7.1 Recursive Descent Parsers

Recursive descent parsers make the dominating share among parsers used in practice [Mat+19]. They operate in a top-down manner, recursively calling parser subroutines that match non-terminals, and can precisely parse $LL(1)$ grammars [Knu71] whose production rules are unambiguous when reading a string character by character. Having a context-free grammar, it is possible to generate sound recursive descent parsers automatically, for instance, with ANTLR [PQ95].

Listing 2.3 shows a recursive descent parser written in C for the parenthesis language. The single production rule is called recursively. Thus, the parser does not require loops for parsing, while it still can parse strings of arbitrary length (as long as the stack memory is not exhausted during execution).

2.7.2 Fuzzing with Grammars

Grammars can be used to generate structured random inputs for fuzzing [SP21; HZ19], which have much higher chances of triggering code beyond the first input parsing stage in the

Listing 2.3: A recursive decent parser for correctly parenthesized expressions

```c
void parse_S(const char *input, int *position) {
  if (input[*position] == '(') {
      (*position)++;
      parse_S(input, position);
      if (input[*position] != ')')
          error("Expected␣closing␣parenthesis");
      (*position)++;
      parse_S(input, position);
  }
}
```

target program. The advantage is that grammar-based fuzzing does not require a coverage feedback mechanism, but it requires the input specification of the target program in form of a grammar. As a result, grammar-based fuzzing is, in particular, interesting for testing embedded systems, where retrieving coverage feedback is hard [Mue+18b; Wri+21; Eis+22]. Generating input from grammars can be done trivially, for instance, by starting at the start symbol and randomly selecting production rules that match non-terminal symbols in the current string until only terminal symbols remain. More sophisticated methods measure the coverage of applied production rules and try to maintain an equal distribution across all possible paths [HZ19].

3 Embedded Fuzzing – State of the Art

In this chapter we explain challenges regarding embedded fuzzing and systematically present methods that try to overcome these. We contend that the diversity of embedded systems on multiple levels is the essential reason why fuzzing embedded systems is still an open challenge at present. Reliable, holistic fuzz testing of embedded systems ought to cover both the firmware code and the appropriate environment for that firmware. Due to diversity the fuzzer needs to scale up to innumerable variants of hardware and firmware that are often poorly documented.

We hypothesize that a golden tool and solution for fuzzing embedded systems (*embedded fuzzing* for short) does not exist yet. To verify this hypothesis, we formulate the following research question: *What are the main features and limitations of current tools for fuzzing embedded systems?* To address this question, this chapter conducts a systematic review of the state of the art of approaches to embedded fuzzing. Our review rests on a formal description of fuzzing for embedded systems and our developed taxonomy used to categorize the reviewed works. We leverage these to advance a clustering of the reviewed works upon the basis of their underlying mechanisms.

The treatment highlights that emulation-based approaches work well for academic examples but may fail on real-world use cases. By contrast, hardware-based approaches with all their incarnations may yield good results though are less scalable and portable. Hybrid approaches seem to bear disadvantages from both worlds. By presenting the whole picture of fuzzing for embedded systems, this chapter demonstrates features as well as limitations of each reviewed work, ultimately demonstrating what kind of future research is needed and deriving directions on how to pursue it.

Section 3.1 defines the criteria for a piece of research to be included in our review, and Section 3.2 introduces our extended model for fuzzing embedded systems. Thereafter, we review related work of hardware-based and emulation-based embedded fuzzing in Section 3.3 and Section 3.4, respectively. Abstraction-based approaches are reviewed in Section 3.5. We summarize the relevant works for embedded fuzzing in Section 3.6, discuss future trends in Section 3.7, and related work in Section 3.8. We conclude the chapter in Section 3.9.

3.1 Inclusion Criteria

To systematically find relevant work on embedded fuzzing we first define our *inclusion criteria* (C1 - C4) in this section, as follows:

C1 Research papers that are published in the top five venues in the category “Engineering & Computer Science”, sub-category “Computer Security & Cryptography” according to Google Scholar [Goo22].

C2 Research papers that are published during the five years between 2017 and 2021.

C3 Research papers that mention “fuzzing” and “firmware” or, alternatively, “fuzzing” and “embedded”.

C4 Research papers or tools that we feel convey relevant approaches to embedded fuzzing.

The first two criteria are objective, as Scholar offers convenient selection and sorting facilities for research venues. The chosen area of security is the one that we found most relevant to fuzzing in general, considering fuzzing as a technique for unveiling software vulnerabilities that an attacker could exploit. To confirm this, we also tried subcategories “Software Systems” and “Computing Systems” but none of the corresponding papers survived the criterion C4. The five venues arising through the first criterion are:

V1 ACM Symposium on Computer and Communications Security.

V2 IEEE Transactions on Information Forensics and Security.

V3 USENIX Security Symposium.

V4 IEEE Symposium on Security and Privacy.

V5 Network and Distributed System Security Symposium.

Criterion C3 is also objective. Scholar offers a convenient search facility for the contents of published papers. We searched in each of the five identified venues with following search string:

```
fuzzing AND (firmware OR embedded)
```

However, many papers identified this way were not relevant to our purposes for a variety of reasons, ranging from fuzzing being treated only marginally or being mentioned only in the paper references. Here is where criterion C4 comes into play, indicating that we had to exercise manual scrutiny to further select the very contributions that would convey relevant approaches and tools for embedded fuzzing.

Moreover, we decided to appeal to an additional, purposely subjective, inclusion criterion in order to freely represent our experience through the review. It is apparent that criterion C4 does not deliberately refer to a specific time window or venue, hence applying it in isolation from the previous criteria provides us with the freedom of selection we also wanted to have. Therefore, our resulting inclusion criteria can be represented as a sentence in propositional logic:

$$(C1 \land C2 \land C3 \land C4) \lor (C4 \land \neg(C1 \land C2 \land C3)).$$

Clearly, this sentence is logically equivalent to C4 because our personal judgement had to be applied to all possible candidates. However, its construction allows us to represent the numbers of papers for the meaningful combinations of criteria and venue as well as the papers that we freely decide to consider. Such numbers, in particular for the two main disjuncts in the sentence, can be found in Table 3.2. The selection process is additionally depicted in Figure 3.1.

It can be understood why our review features a total of 42 papers.

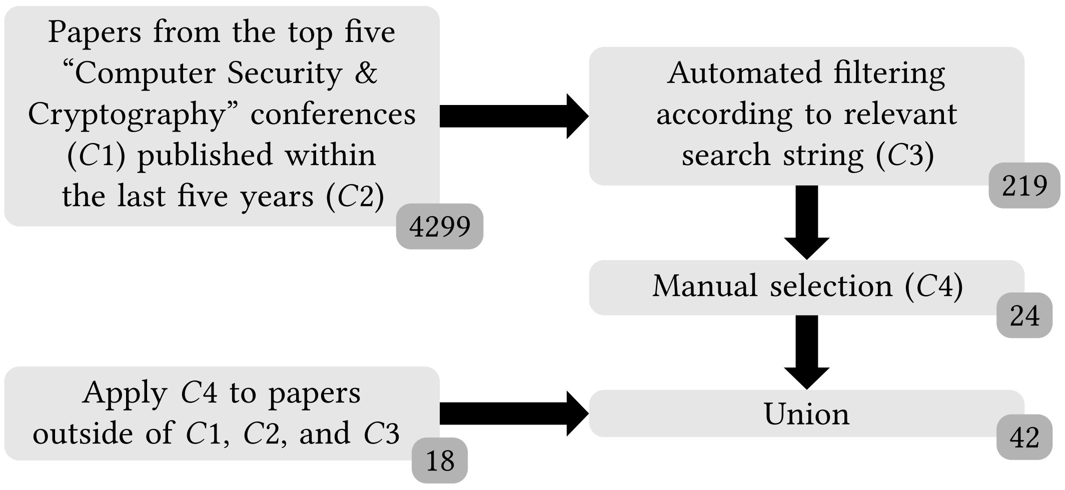

Papers from the top five "Computer Security & Cryptography" conferences (C1) published within the last five years (C2)
4299
Automated filtering according to relevant search string (C3)
219
Manual selection (C4)
24
Apply C4 to papers outside of C1, C2, and C3
18
Union
42

Figure 3.1: The selection process for finding relevant works, including the numbers of papers each step has mined.

3.2 Terminology

In this section, we propose a formal description of embedded fuzzing to mathematically describe fuzzing as a stochastic process. We therefore describe the distinct tasks an embedded fuzzer must fulfil in an algorithmic manner. We use the notation introduced by Böhme [Böh18] and apply it to fuzzing *systems*.

Let a system $\mathcal{S}$ be our target that we fuzz. The sample space for system $\mathcal{S}$ is the *input space* $\mathcal{D}$. *Fuzzing* is then a stochastic process $(\mathcal{D}, \mathcal{F}, P)$ of selecting inputs t_i from the input space $\mathcal{D}$. The event space $\mathcal{F}$, or *fuzzing campaign*, is then the collection of all drawn input, i.e.

Definition 10: *[Fuzzing Campaign]*

$$\mathcal{F} = \{t_i | t_i \in \mathcal{D}\}_{i=1}^{N}$$

The probability function P dictates the selection of an input t_i with probability p_i to be part of the fuzzing campaign $\mathcal{F}$. Note that we leave out the often used but poorly specified terms blackbox, graybox, or whitebox fuzzing. The degree of *smartness* is modeled by adjusting probability function P, i.e. probability p_i for each drawn test input. A tool that implements the sampling function of $(\mathcal{D}, \mathcal{F}, P)$ is called a *fuzzer*.

The probability function P can depend on observations of the system $\mathcal{S}$. If no observations influence the probability p_i for selecting a new input t_i (all p_i's are equal), the *fuzzing campaign* is a uniform random tester[1].

Sampled inputs t_i are processed by system $\mathcal{S}$ with its configuration $\mathfrak{C}$, as in Definition 11. The configuration $\mathfrak{C}$ describes the static environment of the system, including hardware properties.

[1]Even a non-deterministic blackbox fuzzer could have some non-empty observations or some non-uniform probabilities.

Table 3.2: Numbers of papers per criterion and venue.

	C1∧C2	C1∧C2∧C3	C1∧C2∧C3∧C4	C4 ∧¬(C1∧C2∧C3)
V1	1400	61	2	-
V2	1350	12	1	-
V3	716	79	15	-
V4	518	38	2	-
V5	315	29	4	-
Σ	4299	219	24	18

In contrast to existing formal definitions, we introduce an observing mechanism that can observe system $\mathcal{S}$ in desired dimensions that are not further specified. The observation of the system's behavior when processing input t_i is then described by $O_{t_i} \in \mathcal{O}$ and is obtained by

Definition 11: *[Execution Observation]*

$$O_{t_i} \xleftarrow{observe} \mathcal{S}_{\mathfrak{C}}(t_i), 1 \le i \le N$$

where $\xleftarrow{observe}$ describes the observations of the system during the execution. This construction allows, for example, to gather code coverage of a system or to observe whether exceptional states of the system have been reached. It also allows us to monitor emitted physical side-channel data or perform liveness checks of the system after a processed input. Further observations can be execution time or the output of a system. The specific observation space depends on the actual device and observer.

For fuzzing, Algorithm 1 is built around Definition 11, which is called in line 4, where O_{t_i} is the concrete observation of system $\mathcal{S}_{\mathfrak{C}}$ on processing input t_i.

The algorithm continuously samples inputs $t_i \in \mathcal{D}$ on behalf of the probability function P, which are then processed by system S. The observation O_{t_i} is inspected for *unspecified behavior* in function SPECIFIED. For example, the specification can contain maximum execution durations or illegal states of the system. If unspecified behavior is discovered, the (hopefully) responsible input t_i is preserved in $T_{\times}$.

Finally, the probability function P may be adjusted by function ADJUST, based on the new observation O_{t_i}. For example, mutation-based coverage-guided fuzzers implicitly alter their probability function, when a new execution path has been discovered by adding the responsible input to an input corpus. On each iteration, a seed is picked from the input corpus and mutated randomly to generate a new input – so the seeds directly influence the probability space of newly sampled inputs.

Algorithm 1: System fuzzing algorithm.

```
    Input: System $\mathcal{S}$ with configuration $\mathfrak{C}$, initial seed corpus $\mathbb{C}$, probability function $P$
    Output: Inputs leading to unspecified behavior $T_\times$
1 $T_\times = \varnothing$ while $\neg($ TIMEOUT() $\vee$ ABORT()) do          // fuzzing loop
2     Pick $t_i \in \mathcal{D}$ with probability $p_i$                      // sample input
3     $O_{t_i} \xleftarrow{observe} \mathcal{S}_{\mathfrak{C}}(t_i)$
4     if $\neg$ SPECIFIED($O_{t_i}$) then
5         $T_\times = \{t_i\} \cup T_\times$                                // preserve input
6     end
7     $P$ = ADJUST($P$,$O_{t_i}$)                                        // may benefit from $O_{t_i}$
8 end
```

Differential Fuzzing [NNP19; Nol+20; HEC20] refers to fuzzing of functionally equivalent programs with the goal of revealing differences in observations O_{t_i}, particularly diverging outputs of the programs which indicate logical bugs. With an adaption of algorithm 1, systems can be fuzzed differentially, e.g. to test two implementations of the same algorithm for a deviating behavior.

We model *stateful* fuzzing by allowing t_i to contain multiple inputs, $t_i = \langle t_i^1, t_i^2, \dots, t_i^m \rangle$. Executing such a sequence on system $\mathcal{S}$ brings it to a state s, which we collect as part of $\mathcal{S}$'s observation O_{t_i}.

Ensemble Fuzzing, as introduced by Chen *et al.* [Che+19], is when multiple fuzzers execute algorithm 1. The main idea is that the different tools synchronize their observations. The same system S can be run with different configurations $\mathfrak{C}$ and $\mathfrak{C}'$. For example, configuration $\mathfrak{C}'$ can have the input validation, such as a checksum, turned off to allow a fuzzer to get deeper into the System under Test (SUT) more quickly. The original configuration $\mathfrak{C}$ is then used to validate inputs from configuration $\mathfrak{C}'$ to reduce false positives.

Fuzzing Harness, or *Fuzz Wrapper*, is an adapter between a fuzzer and a specific target. Applications that process data directly from a file or console input channel can most likely be fuzzed without any adapter in between. For all other cases – a typically lightweight – fuzzing harness is necessary to route input data from the fuzzer to the target's interface.

3.3 Hardware-Based Embedded Fuzzing

The high coherency of software and hardware in embedded systems suggests that fuzz testing is to be performed on the actual device. However, observing of the device, i.e. implementing $\xleftarrow{observe}$, already poses a challenge. In this section, we present approaches that aim to run the target application in its designed hardware environment.

Fan *et al.* [FPH20] ported the popular fuzzer AFL to ARM-based Internet of Things (IoT) devices. Within their ARM-AFL project they developed a code instrumentation strategy for ARM assembly and implemented a lightweight heap memory corruption detector. The whole fuzzing process runs on the target device itself, leading to a high throughput. In principle, the fuzzing process works exactly like fuzzing on a desktop PC. The target process is observed on crash signals and code coverage in each O_{t_i}. ARM-AFL requires Linux as the operating system and the source code of the target program.

Frida [Rav19] is a dynamic code instrumentation toolkit that can hook into arbitrary user processes enabling transparent access to the execution. It can also be controlled remotely, allowing for hooking into Linux, QNX, Android, and iOS applications. In addition, Frida enables the collection of code coverage data from the hooked process to facilitate fuzzing. However, the Frida server application must be executed on the target device, which can be challenging on closed/commercial devices.

Bogdad and Huber [BH19] developed Harzer Roller – a linker-based instrumentation tool for embedded security testing. They address the problem that embedded firmware often needs closed-source libraries in order to communicate with the hardware, which cannot be instrumented by the compiler. These libraries are usually shipped as an object file and are integrated into the firmware by the linker. To be able to generate call traces, all functions within the object file are renamed, and appropriate proxy functions are generated. For detecting stack overflows, a stack canary can be generated by the framework before calling the original function. The authors state that this technique is meant for simple embedded devices with limited debug capabilities. The instrumentation of an object file increases its size up to 150%, which usually makes it impossible to instrument all libraries on memory-limited targets. The framework has been used for fuzzing an ESP8266 using Boofuzz [Per17] as blackbox fuzzer.

Oh *et al.* [Oh+15] present a simple Dynamic Binary Instrumentation (DBI) method for embedded systems without any dependency on the operating system. They connect the target device with a debugger and insert software breakpoints at manually chosen locations. When a breakpoint is reached, the instrumentation framework is notified, and the breakpoint is removed for further execution. This method enables observation of manually selected, executed code parts in O_{t_i} and could be used for coverage-guided fuzzing of any embedded system that provides a suitable debugger. According to the measurements of the authors, the overhead of this method is only around 1%. However, the measurements have only been performed on one device.

Börsig *et al.* [Bör+20] present a method to instrument code for ESP32 microcontrollers, whereby the coverage data is returned to the fuzzer's host via a JTAG connection. For this, the source code must be available and the GCC coverage instrumentation mechanism is used. The input data is sent to the target via the original channel, e.g. Wi-Fi. However, the transfer of the coverage data via the JTAG interface slows down the fuzzing process roughly by a factor of ten.

Tychalas *et al.* [TBM21] investigate security evaluation of Programmable Logic Controllers (PLCs). Although, PLC binaries are not regular programs, the authors show that they can introduce vulnerabilities into systems. To reveal such vulnerabilities, they propose a method to instrument PLC binaries, and enable coverage-guided fuzzing on them.

Song *et al.* [Son+19] presented PERISCOPE to examine communication between peripherals and drivers over Memory-Mapped IO (MMIO) and Direct Memory Access (DMA) for Linux-based systems. It therefore needs to be compiled into the target's kernel to hook into MMIO and DMA regions that drivers use to communicate with the hardware. The extension PERIFUZZ allows fuzzing on this hardware-OS boundary by injecting fuzzing data into the corresponding memory regions.

Delshadtehrani *et al.* [Del+20] designed the programmable hardware monitor PHMon for debugging, assisting vulnerability detection, and enforcing security policies. A prototype of the hardware monitor has been deployed on an Field Programmable Gate Array (FPGA) in conjunction with a RISC-V processor. It can be used to generate coverage feedback directly from the execution on the hardware. The authors state that coverage-guided fuzzing with PHMon and AFL is 16 times faster than fuzzing in a full-system emulator. However, the hardware monitor module needs to be included directly on the hardware chip, to enable this performance advantage.

Sperl *et al.* [SB19] present a side-channel approach of gathering code coverage from embedded systems by precisely monitoring the power consumption of the target device during execution. Therefore, an oscilloscope is used to record power traces, which are processed further on a host PC to recognize the different executed basic blocks. The recognition is realized by machine learning classification algorithms. With this technique, they are able to approximate the control flow graph with correlation coefficients of up to 0.9. For correct results the setup needs to be calibrated and trained on the actual device under test.

García *et al.* [Gar+20] use timing and electromagnetic emanation side channels from embedded devices for analyzing implementations of cryptographic algorithms. They use these side channels in a specialized feedback-driven fuzzing algorithm to recover cryptographic private keys.

Chen *et al.* [Che+18a] present IoTFuzzer, which aims for fuzzing IoT devices that are controlled by mobile phone applications – in this case Android apps only. It makes use of the fact that accompanying mobile apps of IoT devices are aware of the exact protocol and encryption for controlling the device. The idea is to reuse the mobile app to send correct messages to the target device, thereby enabling protocol-aware fuzzing. For this, the mobile app is initially scanned for functions that consume user input and send it to the IoT device. These functions are then re-used to send fuzzing messages to the target device. This way, the generation of syntactically and semantically correct fuzzing messages is ensured. Crashes are detected by observing the communication or performing liveness checks.

Redini *et al.* [Red+21] have refined this method in their tool DIANE. In contrast to IoTFuzzer, DIANE tries not to hook into the function that consumes user input first, but the last possible one, before the message is encoded and send to the SUT. Thereby, eventual sanitization of the user input within the mobile application is bypassed and the possible input space is enlarged.

Snipuzz [Fen+21], also aims to fuzz test IoT devices with accompanying mobile applications. Unlike IoTFuzzer and DIANE, it additionally analyzes responses from the target device to enable feedback-driven fuzzing. Appropriate message sequences are gathered by reading the public API, when it is available, or from analyzing the communication between the accompanying mobile application and the target device. As an alternative, the accompanying mobile application can also be disassembled, but this usually requires more effort. Although Snipuzz aims to be lightweight, it requires some manual analysis to gather valid initial seeds and select the right message sequences for fuzzing.

Aafer *et al.* [Aaf+21] present a technique to perform feedback-driven fuzzing of Android TV boxes based on logging outputs. First, static analysis is applied to extract logging statements within the target's firmware. With taint analysis, the collected logging statements are classified according to whether they are related to input validation. This labelled collection of logging statements is then used to train a neural network model, which serves as a classifier for logging outputs. During fuzzing, output logs are analyzed by using the model to detect diverging behavior of the target and to provide feedback to the fuzzer. In addition, they introduce an external component that detects visual and auditory anomalies by capturing and comparing video and audio signals before and after each fuzzing step. This method generates a coarse-grained feedback, compared to branch code coverage, and is designated for rather talkative devices, that give feedback via logs.

3.4 Emulation-Based Embedded Fuzzing

Emulators offer transparency and control of the emulated subject and enable a precise observation O_{t_i} of internal operations in manifold dimensions. Furthermore, multiple instances of an emulator can be created easily, enabling horizontal scaling of the fuzzing process.

However, running firmware of embedded devices in an emulator is subject to several challenges, which are carved out well by Wright *et al.* [Wri+21]. Most notable for fuzzing is the fidelity and the effort needed to adapt an emulator to a specific target.

Executing the application within an emulator can be realized by either replacing the hardware layer with a system emulator or by moving only the application into a user-mode emulator.

This section presents the most notable approaches that enable embedded fuzzing in an emulator.

3.4.1 User Mode Emulation Fuzzing

User applications, built for running in an operating system, can potentially be executed very easily in an emulator, because of the well-defined operating system interfaces at the application layer. An emulator therefore replaces or interchanges the operating system layer i.e. the kernel. User mode emulators like QEMU can thereby even emulate applications from (in particular Linux-based) embedded systems independently of the instruction set architecture. In best case only system calls from the application must be forwarded to the host system. However, applications might expect specific hardware (drivers) or file systems to be present and the emulator needs to treat according requests adequately.

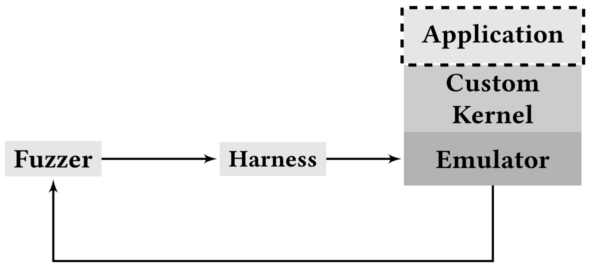


Figure 3.3: Scheme of fuzzing applications in a user-mode emulator.

All investigated fuzzing frameworks in this category use a custom kernel for this purpose, also depicted in Figure 3.3. The dotted box depict that only the application layer originates from the actual target firmware.

Chen *et al.* [Che+16] developed the Firmadyne framework, which allows for automated dynamic analysis of Linux-based embedded firmware images. It extracts the root file system from a binary firmware image and utilizes a custom kernel to run the image within the QEMU full-system emulator. With this setup, dynamic analysis of the user applications in the firmware can be performed, which is demonstrated by providing a set of known exploits that can be tried on the emulated device. Even though the full-system mode of QEMU is used, Firmadyne should be considered to enter at the application layer, because it deploys its own customized kernel and only the user space applications from the firmware are executed. The custom kernel partially compensates for missing hardware emulation, for example, by providing an emulated NVRAM that embedded devices often use.

The Firmadyne framework is enhanced by Kim *et al.* in FirmAE [Kim+20]. They claim that the Firmadyne framework could only get 16.28% of their tested set of firmware images up and running for dynamic analysis. To solve this problem, they introduced heuristics to configure boot parameters, kernel parameters, network interfaces, and file systems correctly. With these modifications, they were able to automatically run 79.36% of the aforementioned set of firmware images within QEMU.

FirmFuzz [Sri+19] is an automated introspection and analysis framework for IoT firmware. It is designed for embedded devices that offer user interfaces through a webpage and are based on Linux. The QEMU system emulator is set up with a customized kernel in conjunction with fake peripheral drivers to compensate for potential missing hardware emulation. A headless browser is used to communicate with the device automatically through a virtual network interface to find user interfaces. After the static analysis of the firmware, a generation-based fuzzer is set up. Seed input data is generated, using the contextual information that is gathered from the firmware image. The target is monitored for faults by the modified Linux kernel within the emulator.

FIRM-AFL [Zhe+19] is based on AFL and Firmadyne. The idea is to speed up fuzzing within QEMU by letting the target user process run in the user-mode as long as possible. When necessary, the user process is translated to the full-system emulator of the appropriate device hardware. As a result, the overhead of a full-system emulation is largely omitted. The authors state that with this mechanism, the fuzzing process can be sped up by a factor of ten. However, it is required that the target device runs a POSIX-compatible operating system and the hardware can be emulated by QEMU.

Transferring embedded applications from Linux-based devices into an emulator by providing a customized kernel can be successful in some cases, in particular when the target application does not rely on special hardware peripherals. Nevertheless, there remain many embedded systems to which this does not apply, and which demand a different approach for emulation-based fuzzing.

3.4.2 Full-System Emulation Fuzzing

Once an embedded system can be emulated adequately, code coverage, fault states, and other meta information of the execution can be obtained easily. The next section is about methods that enable full-system emulation of embedded devices. For a correct emulation of embedded firmware, all hardware peripheral accesses must be treated in the emulator.

Peripheral Emulation

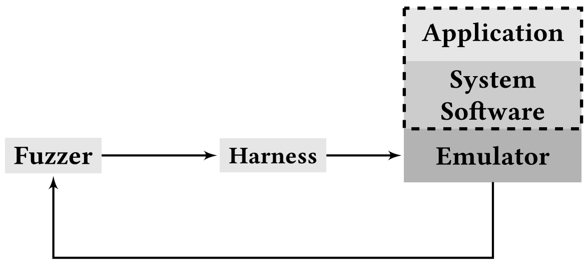


Figure 3.4: Scheme of fuzzing embedded applications in a full-system emulator.

A hardware access manifests itself in read and write operations on the hardware address space. Additionally, hardware interrupts are a mechanism to let hardware peripherals trigger code areas from the firmware. Implementing software equivalents of hardware peripherals and providing them on their expected locations in the hardware address space is a way to enable emulation. When all peripherals from a target device can be emulated, an unmodified firmware image can be executed and fuzzing can be enabled with little effort, as depicted in Figure 3.4.

The QEMU system mode is a popular full-system emulator, which already provides configurations for several microcontrollers and peripherals and supports a large variety of architectures. TriforceAFL [HN16] combines AFL with QEMU and enables emulation-based coverage-guided fuzzing for targets that can be emulated with QEMU. If the desired target device is not supported, the implementation and configuration can be very laborious and requires deep knowledge of the hardware.

Herdt *et al.* [Her+20] present a different solution for emulating the whole hardware of an embedded system. They apply libFuzzer to a SystemC Virtual Prototype (VP). SystemC is defined as IEEE-1666 standard [Gro11] and provides a set of C++ libraries to define virtual prototypes. Virtual prototypes are models of the entire hardware system and allow an accurate simulation. They are an established way of testing systems during their development in the industry. Fuzzing is performed on the virtual hardware by using a fully booted state of the system, which is preserved by a fork-server mechanism. However, the complete system must be described in SystemC, which requires deep insights into the SUT and can again require a lot of manual work.

Clements *et al.* [Cle+20] present HALucinator to address the problem of emulating peripherals by using the HAL as an entry point. First, it locates HAL functions in the firmware through binary analysis. Second, it intercepts the execution of the HAL functions and instead mimics its expected behavior. Handlers for each HAL function must be implemented manually once.

Beside correct emulation, HALucinator can intercept functions that provide random values and is able to replace them by deterministic functions, which can render fuzzing more efficient.

Kim *et al.* [Kim+19] proposed RVFuzzer for detecting *input validation bugs* in robotic vehicles. Robotic vehicles are cyber-physical systems managed in real-time by a microcontroller. It needs to control actuators, process sensor data, and react to control commands. A careful validation of incoming control commands is therefore required, especially if they are received from an unencrypted broadcast medium. RVFuzzer tries to detect (sequences of) control commands that bring the robotic vehicle into an unstable state. Therefore, the control program is connected to a physical simulation of the robotic vehicle, and input commands as well as environment parameters are mutated. Instabilities are detected by observing whether the presumed state in the control program deviates too much from that in the simulation.

Peripheral Proxying

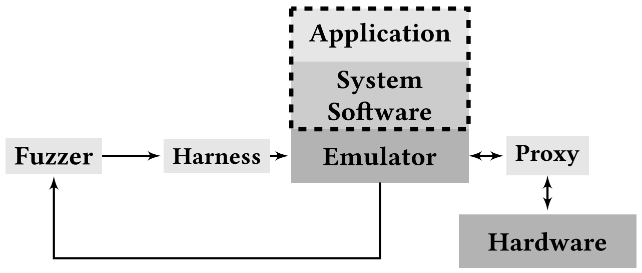


Figure 3.5: Scheme of embedded fuzzing with peripheral proxying.

When deep knowledge about the SUT is missing, hardware accesses of the firmware must be treated differently. An alternative solution is to forward each hardware access to the real device. Therefore, a proxy application is introduced to route appropriate values and triggered interrupts between the actual hardware and the emulation, as shown in Figure 3.5.

PROSPECT [KPK14] uses TCP/IP connection to forward hardware accesses, Avatar [Zad+14] a debugging connection, and SURROGATES [KKM15] routes hardware accesses through a dedicated FPGA to the actual hardware.

Regarding mobile system drivers, Talebi *et al.* [Tal+18] developed Charm that enables fuzzing of device drivers by forwarding hardware peripheral accesses through a USB-based connection. Since the drivers need to be modified for this method, Charm works only with open source drivers.

Avatar has a successor, Avatar^2 [Mue+18a], which is not only intended for hardware access rerouting, but more for orchestrating different frameworks to enable dynamic analysis. Muench *et al.* [Mue+18b] enable coverage-guided fuzzing on a wide variety of devices by using PANDA [Dol+15] as the emulator, Avatar^2 [Mue+18a] for forwarding non-emulatable hardware accesses, and Boofuzz [Per17] as the fuzzer. Furthermore, they uncover the issue of silent memory corruptions that can occur in embedded devices without Memory Management Units (MMUs) or operating systems that take care of memory accesses. These are memory corruptions that do not result in a crash of the device upon occurrence and are therefore are not easily observable. To detect silent memory corruptions, they present heuristics that can be applied to an emulator, regardless of the manner of hardware access treatment. When using these heuristics, all occurring memory corruptions of a device can be discovered.

Peripheral proxying offers a solution for emulating an embedded device without excessive implementation effort. However, the forwarding of peripheral accesses to the real hardware can present a bottleneck, depending on the number of requests to the hardware. Additionally, manual configuration and setup of the proxying mechanism is required.

Peripheral Modeling

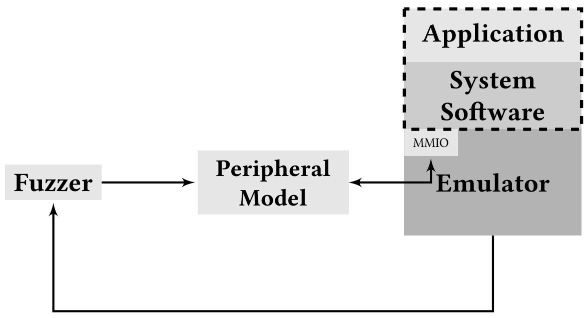


Figure 3.6: Scheme of embedded fuzzing with peripheral modeling.

Where implementing virtual hardware requires too much effort and peripheral proxying is too slow for fuzzing, automated hardware modeling can be a solution. The idea is to learn how to respond to hardware accesses such that the firmware continues its execution. The peripheral model is thereby directly connected to the MMIO address space and can be supported by the fuzzer, as depicted in Figure 3.6.

Gustafson *et al.* [Gus+19] present a semi-automated re-hosting framework, called PRETENDER. They solve the modeling of hardware peripherals by means of preliminary observation and recording of the behavior of the real device with AVATAR[2]. As a result, not only accesses to the hardware are recorded, but also the timings and orders of interrupts. Next, a rather complex step of categorizing MMIO registers and initializing *State Approximation model* occurs. This should allow for smart responses to hardware accesses of the firmware. Finally, human interaction is needed to define the entry point of the fuzzing data. The authors state that PRETENDER allows for a *survivable execution*, which can just be sufficient for a dynamic analysis of the device.

Spensky *et al.* refined this approach with CONWARE [Spe+21], which can also learn hardware peripheral behavior by first recording interactions between the firmware and the real hardware peripheral and subsequently extracting models for each of them. The extracted models can then be used for a full-system emulation. In contrast to PRETENDER, CONWARE claims to be more generic and can even model peripheral behavior that has not been recorded directly.

A hardware-agnostic approach for embedded fuzzing is presented by Feng *et al.* [FML20]. Their framework P^2IM responds to each peripheral access (a read from the MMIO address space) with input data from the fuzzer. Therefore, the MMIO registers are categorized into *Control*

Registers, Status Registers, Data Registers, and *Control-Status Registers* by observing how the firmware accesses the registers. Depending on the category, interaction with the registers is treated differently. Most important is the treatment of *Data Registers*, where P^{2}IM directly injects input data from the fuzzer. Thereby, the fuzzer itself models all the peripheral input generically, omitting the need for finding and choosing the correct input vector for the target. The interrupt emulation is implemented quite pragmatically by sequentially firing one interrupt per 1000 executed basic blocks. When the initially supplied fuzz input buffer is exhausted, the execution is terminated and the code coverage is fed back to the fuzzer. The explorative nature of the fuzzer is used to improve the hardware peripheral modeling successively. The framework allows existing fuzzers to be added as a drop-in component, offering AFL as default. However, peripherals that use DMA are not modeled by P^{2}IM, as this would require insights on the internal design of the target device.

For automatic emulation of DMA input channels in P^{2}IM, Mera *et al.* [Mer+21] present the drop-in solution DICE. It observes the behavior of running firmware in the emulator and recognizes candidates for DMA input channels heuristically. In principle, it searches for pointers to the internal RAM that are written to memory-mapped IO-registers. The authors claim that, during their tests, DICE did not create any false positive categorization and successfully detected 21 out of 22 actively used DMA input channels. With negligible overhead, it enables fuzzing of DMA input processing firmwares without further hardware knowledge.

Johnson *et al.* [Joh+21] present a more *targeted* peripheral modeling approach with JETSET. In this case, an analyst manually defines a goal address in the firmware that should be reached, and JETSET tries to derive the necessary hardware peripheral responses to reach this address with symbolic execution. For instance, the transition from kernel space to user space can be used as such a goal address. The explicit goal address allows JETSET to mitigate path explosion during symbolic execution.

Zhou *et al.* [Zho+21] enable peripheral modeling in their tool µEMU by mixing symbolic and concrete execution to calculate appropriate responses to hardware accesses. First, all hardware peripheral dependent inputs are treated symbolically. To avoid path explosion, symbolically calculated values are cached and reused during concrete execution. When invalid execution states are reached, the responsible cached values and the state itself are marked as invalid and different paths are taken by future symbolic executions. This way, the hardware peripherals are enhanced iteratively.

Scharnowski *et al.* [Sch+22a] refine the mechanism of P^{2}IM. Instead of putting a memory-mapped register into a category, their framework FUZZWARE handles each individual access to a memory-mapped register by additionally considering the program counter on each access. On the first occurrence of an access, the emulator is reset to the instruction right before accessing the memory-mapped register and symbolic execution is used to determine whether and how the value affects the further execution. Accordingly, the individual memory-mapped register access is assigned just enough random input bits to ensure that all dependent branches can be reached. This leads to a minimal consumption of input bits from the fuzzer while fuzzing the whole peripheral interaction. The authors claim that DMA could also be modeled with further effort, but this is considered out of scope of their work.

3.4.3 Sandbox Emulation Fuzzing

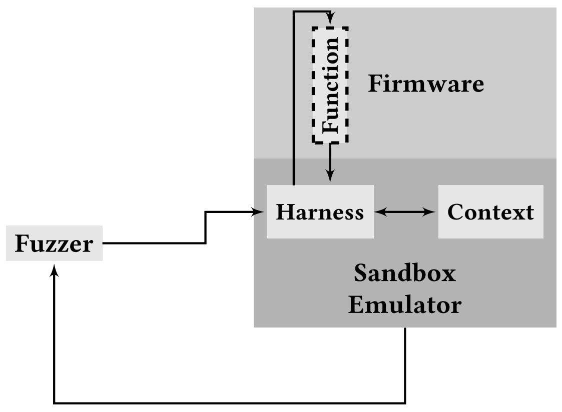


Figure 3.7: Scheme of embedded fuzzing through sandbox emulation.

In cases where a full-system emulation is not feasible, lightweight sandbox emulation can be a solution. Thereby, the binary code is executed from a manually chosen point with a manually created context. The idea is to fuzz functions that do not communicate with peripherals at all, meaning that the hardware peripherals do not need to be emulated. This technique is almost hardware-independent since only a simulator for the respective instruction set is required. Fuzzing a function from a binary firmware file within a sandbox can be realized as shown in Figure 3.7.

Miasm is a reverse engineering tool to analyze, modify, and partially emulate binary programs. It offers features such as assembling and disassembling for various architectures, emulation with Just-in-Time (JIT) and symbolic execution. In combination with Python-AFL, Miasm can be used to perform fuzzing [Gue17]. Therefore, a sandbox is created by Miasm, input data needs to be mapped to appropriate memory addresses, and registers need to be initialized correctly. This technique is mainly interesting for penetration testers, who reverse engineer binaries and want to perform fuzzing of interesting functions in this way. If the source code is available, it is easier to perform fuzzing of hardware-independent functions by compiling them into a user application and using a general purpose fuzzer.

The Unicorn CPU Simulator [ND15] is used by Nathan [Vos17] similarly.

Maier *et al.* present BaseSAFE in [MSP20], where they also used the Unicorn CPU Simulator to fuzz different layers of a smartphone baseband chip on manually selected target functions and manually created memory contexts. The downside of these sandbox emulation fuzzing approaches is the constrained, manual selection of the target function and manual creation of the execution context.

A semi-automated approach of supplying an execution context to the target code is presented by Harrison *et al.* [Har+20] with their tool PartEMU. They present required steps that allow experts to set up and configure an emulator to enable dynamic analysis of *trust zones* from embedded systems. Based on a set of criteria, hardware and software components are either emulated or reused, and they explain how specific emulation stubs can be implemented. Nevertheless, developing such an emulation-based execution context can involve huge manual effort and requires expert knowledge.

Ruge *et al.* taking the manual sandboxing to the extreme with their highly specialized framework for fuzzing wireless modem firmware in an emulated environment Frankenstein [Rug+20]. They run the firmware of a Broadcom Bluetooth chip within QEMU user mode. Through sophisticated reverse engineering, about 100 locations in the code have been determined, where the execution needs to be redirected and substituted manually. This hooking is required to ensure correct emulation of the firmware. With this setup, they were able to fuzz the Bluetooth modems of popular mobile phones from Apple and Samsung and unveiled several security problems. However, the setup is highly customized and requires a lot of manual effort to adapt it to other embedded firmware.

An automated sandbox-based fuzzing tool for IoT Firmware is presented by Gui *et al.* with FIRMCORN [Gui+20]. First, the firmware image is disassembled, and detected functions are rated based on the memory operations they contain and the use of predetermined *sensitive functions*, such as `read`, `strcpy`, and `execve`. For high rated functions, a context dump (memory and register values) at the starting point of the function is gathered from the actual device. This allows specific fuzzing of potential vulnerable functions within the CPU emulator Unicorn. An automated mechanism detects crashes of the emulator, which result from missing emulated hardware, and skips these crashing functions during further virtual execution. They state that the tool is developed for Linux-based devices only, but it should be possible to extend it to further platforms.

3.5 Abstraction-Based Execution Environment

Symbolic execution is known for several decades [Kin76] and seems not to be located within the domain of fuzzing at first glance. It analyzes the target program independently of its execution environment. The core idea is to treat all input vectors of a program symbolically (similarly to a variable in a mathematical formula) and derive input constraints for all possible program paths. From these constraints, concrete inputs can be extracted that are known to trigger all possible program paths – which is exactly the goal of fuzzing.

However, for each conditional branch in a program, each possible path must be considered in different states. This can lead to the state explosion problem and usually prevents the use of pure symbolic execution in real-life applications.

3.5.1 Symbolic Execution of Embedded Firmware

Symbolic execution does not execute the program code directly, but rather interprets it. It is therefore a good candidate for tackling the challenge of lacking hardware peripheral emulation. All values from hardware peripherals can therefore be symbolized, and possible program paths can be calculated. However, the more hardware values are symbolized, the more constraints and paths are present, which can easily grow exponentially.

Davidson *et al.* [Dav+13] implemented FIE, which allows symbolic execution of firmware for MSP430 microcontrollers by using a modified version of KLEE [C+08]. They assume that software of embedded systems is *simple* enough to allow symbolic execution. Therefore, the target firmware is compiled into a representation that can be symbolically executed with KLEE. FIE includes two notable optimizations: *state pruning* and *memory smudging*. State pruning detects whether the current state has already been reached before and prunes it, instead of adding it to the set of active states. The memory smudging function allows avoiding an intractable state, e.g. an infinite loop with an increment inside. In this case, the state pruning cannot work because the state is not equivalent due to the presence of the increased variable. The memory smudging sets a threshold for consecutive states that differ only in one memory location.

Corteggiani *et al.* [CCF18] present Inception, a symbolic execution engine for embedded firmwares, also based on the KLEE engine. They added a mechanism to symbolically execute assembly code, which is commonly found in embedded firmware code. Additionally, they enable hardware access forwarding for retrieving concrete values from the actual hardware to reduce the symbolical input space.

3.5.2 Concolic Execution of Embedded Firmware

Concolic execution refers to the combination of **conc**rete and symb**olic** execution. In this case, traces are used to analyze reached conditions during a concrete execution, and related constraints are derived. These constraints can be used to generate new input data that exercises a different path of the code. This idea is also termed as hybrid or concolic fuzzing.

Several general-purpose hybrid fuzzers, such as QSYM [Yun+18], SymCC [PF20] are available, as well as frameworks that focus on concolic execution for embedded firmwares. Herdt *et al.* [Her+19] present an approach to integrate a concolic testing engine based on virtual SystemC prototypes for the RISC-V architecture. This is once again subject to all the requirements of virtual prototypes.

Ai *et al.* [ADG20] propose a concolic execution approach for embedded devices that supports various architectures. They perform the concrete execution on the physical device and move the symbolic execution to the host via a debugging connection.

Although concolic execution is a promising method to test code, it faces similar challenges as other embedded fuzzers, because it relies on concrete program traces.

3.6 Reviewing Embedded Fuzzing Approaches

Before we classify the presented embedded fuzzing approaches, we develop our taxonomy criteria. Devising meaningful categories for the existing approaches in order to effectively group them requires care and consideration of existing attempts. Notably, general principles for evaluating and benchmarking traditional fuzzers exist, as suggested by Klees *et al.* [Kle+18]. Accordingly, fuzzers should be tested against a large set of benchmark programs, such as GCG [Def14] or LAVA-M [Dol+16] multiple times for at least 24 hours, with the performance plotted over time. The performance should ideally be measured in the number of detected bugs. The reached code coverage can be used as a secondary performance measure. Additionally, different sets of seeds should be considered and documented. Arguably, a transfer of these principles to embedded fuzzers would be useful. However, current research on embedded fuzzing still faces more fundamental issues of portability and scalability, namely about enabling a fuzzing approach over the widest possible variety of embedded systems of any complexity.

Wright *et al.* [Wri+21] propose to compare different re-hosting frameworks particularly with regard to the amount of user interaction needed for the setup, termed as application effort. The application effort refers to the ease of adapting a framework to new targets. Preferably, a framework can be adapted with little knowledge of the target and low configuration effort. It could be measured in the estimation of time required for the setup, but this would heavily depend on the developer, thus making the results highly subjective and inappropriate for an objective comparison.

3.6.1 Taxonomy Criteria

In light of the existing classification attempts, we feel that the relatively young field of embedded fuzzing may currently be partitioned most beneficially on the basis of how the execution environment is served to the SUT. Therefore, we build three essential categories: hardware-based approaches for those that use the very hardware of the SUT to operate, emulation-based approaches for those that re-host the firmware of the SUT into an emulator, abstraction-based approaches for those that abstract away the details of the hardware. We further classify each category according to finer observations.

Hardware-based approaches let the target software run in its designated environment. Therefore, we decide to further divide these approaches upon the basis of how they gather feedback from the hardware about the execution of the software. Thus, the hardware category features the three sub-categories Instrumentation, Side-Channel, and Message Interface Reusing.

A defining feature for emulation-based approaches is the way they treat hardware peripheral accesses. Therefore, we coherently decide the five sub-categories User Mode Emulation, Full-System Emulation, Peripheral Proxying, Peripheral Modeling, and Sandboxing.

The last category features abstraction-based approaches, hence the two sub-categories for enabling the abstraction process are Symbolic Execution and Concolic Execution. It should be noted that concolic approaches usually need traces from the execution environment and therefore a concrete execution environment but (manually) selected input vectors can be made symbolic. Therefore, we decide to keep these with abstraction-based approaches.

These categories, also depicted in Figure 3.8, serve as a first criterion for our *taxonomy criteria* and help us to distinguish the general mechanism of each approach. Our defined taxonomy, based on the approaches' execution environments, is again reflected in the *rows* of Table 3.9. Second, the *columns* in Table 3.9 show what we feel are the relevant elements of comparison for each work.

- *Source Code Agnostic* - This criterion indicates whether the fuzzer requires the source code of the SUT to run, which is a major factor for many application scenarios.

- *Available* - This criterion indicates whether any implemented tool of the proposed approach is readily available and functioning, irrespective of whether it is open or closed source.

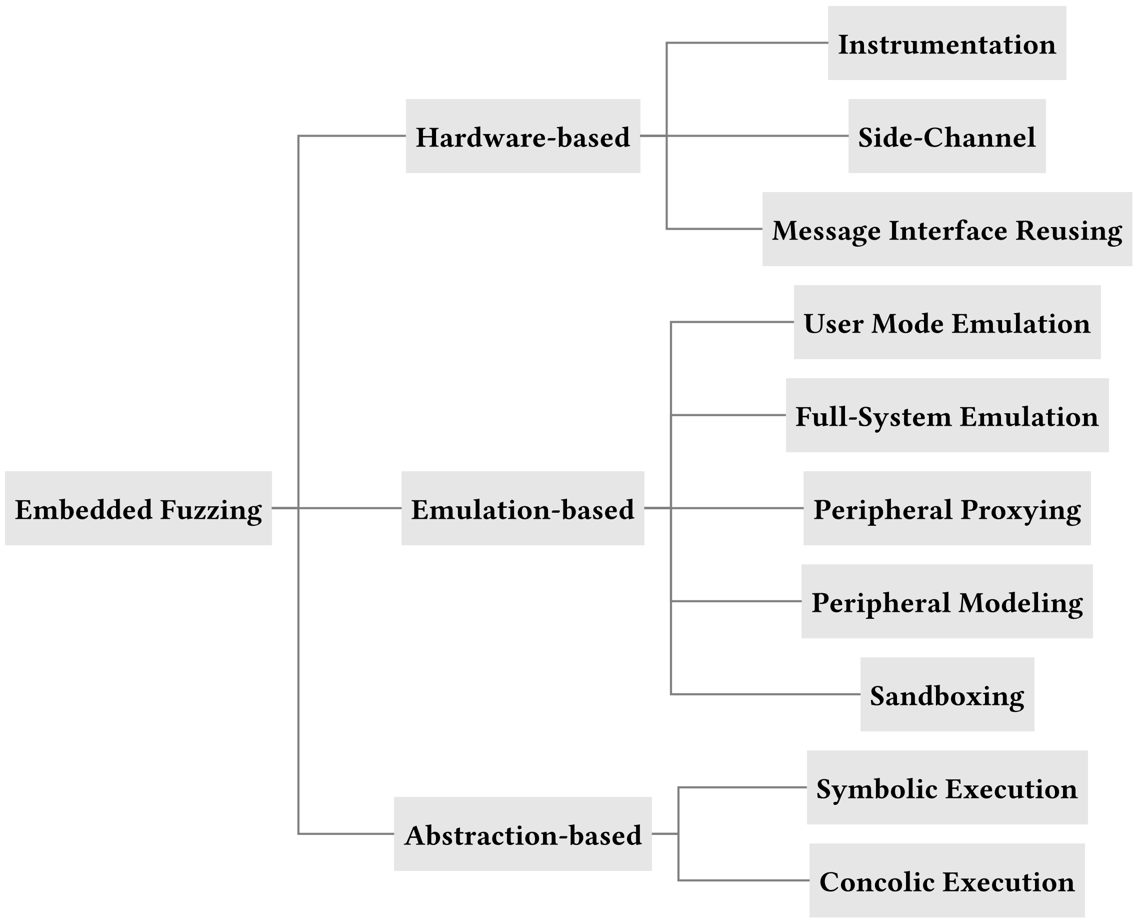


Figure 3.8: Embedded Fuzzing Taxonomy.

- *Key Contributions & Limitations* - This column presents the key features as well as the limitations of each approach.

3.6.2 Overview of Embedded Fuzzing Approaches

A summary of the relevant embedded fuzzing approaches arranged according the criteria from Figure 3.8 is given in Table 3.9.

Table 3.9: Reviewed embedded fuzzing works.

Environment		Tool	Source Code Agnostic	Available	Key Contributions	Limitations
Hardware-based	Instrumentation	ARM-AFL [FPH20]	✗	✗	Static instrumentation for ARM code	On-target fuzzing only
		Frida [Rav19]	✓	✓	Dynamic instrumentation for various OSes	Application on the target required
		Harzer Roller [BH19]	✓	✗	Static instrumentation for object files	Function traces only
		Os-less DBI [Oh+15]	✗	✗	Dynamic instrumentation with breakpoints	Manual selection of breakpoint locations
		ESP32 Fuzzing [Bör+20]	✗	✓	Static instrumentation for ESP32 applications	Slow coverage data transmission
		ICSFuzz [TBM21]	✓	✓	Static instrumentation for PLC binaries	Dedicated to PLCs
		PERIFUZZ [Son+19]	✗	✓	Fuzzing at HW-OS boundary, driver monitoring	Must be compiled into the kernel
		PHMon [Del+20]	✓	✓	Hardware module for gathering coverage data	Specific hardware required
	Side-Channel	Side-Channel Aware Fuzzing [SB19]	✓	✗	Code-coverage derived from power analysis	Calibration needed
		Certified Side Channels [Gar+20]	✓	✗	EM and timing side-channels	For crypto libraries only
	Message Interface Reusing	IoTFuzzer [Che+18a]	✓	✓	Reuse of accompanying mobile applications	Not feedback driven, Android only
		DIANE [Red+21]	✓	✓	Enhanced IoTFuzzer mechanism	Not feedback driven, Android only
		Snipuzz [Fen+21]	✓	✓	Communication analysis for feedback	For unencrypted channels only

		Android TV Fuzzing [Aaf+21]	✓	✗	Using log output for feedback	Detailed logs needed, Android only
Emulation-based	User Mode Emulation	Firmadyne [Che+16]	✓	✓	Custom kernel for emulation	Linux-based applications only
		FirmAE [Kim+20]	✓	✓	Enhanced Firmadyne mechanism	Linux-based applications only
		FirmFuzz [Sri+19]	✓	✓	Fuzzing of IoT configuration webpages	Linux-based applications only
		Firm-AFL [Zhe+19]	✓	✓	Speedup by hybrid user and system emulation	Linux-based applications only
	Full-System Emulation	TriforceAFL [HN16]	✓	✓	Coverage-guided fuzzing with QEMU	Target must be emulatable by QEMU
		SystemC VP Fuzzing [Her+20]	✓	✗	Coverage-guided fuzzing on VP	Virtual prototype required
		HALucinator [Cle+20]	✓	✓	Re-hosting at HAL	Stubs for HALs required
		RVFuzzer [Kim+19]	✓	✗	Fuzzing controller for robotic vehicles	Rich physical simulation required
	Peripheral Proxying	PROSPECT [KPK14]	✓	✗	Peripherals proxying through TCP/IP	Requires pthreads and TCP/IP support on target
		SURROGATES [KKM15]	✓	✗	Proxying through a custom FPGA	JTAG connection required
		Charm [Tal+18]	✗	✓	Proxying through USB	Recompilation needed
		Avatar^2 [Mue+18a]	✓	✓	Flexible, multipurpose orchestration	Debug access to device required
	Peripheral Modeling	PRETENDER [Gus+19]	✓	✓	Peripheral modeling by recording and learning of peripheral behavior	Unseen peripheral behavior not modeled
		Conware [Spe+21]	✓	✓	Additional modeling of unseen peripheral behavior	Program for recording must be executed on the target
		P^{2}IM [FML20]	✓	✓	Peripheral modeling by automated classification of requests	Missing DMA support

		DICE [Mer+21]	✓	✓	Modeling of DMA-based peripherals	DMA buffer size unknown in advance
		Jetset [Joh+21]	✓	✓	Peripheral modeling by symbolic execution and manual guidance	Manual guidance required
		µEmu [Zho+21]	✓	✓	Peripheral modeling by concolic execution	Caching can cause false hardware models
		Fuzzware [Sch+22a]	✓	✓	Peripheral modeling by detailed classification	Not for complex systems
	Sandboxing	MIASM [Gue17]	✓	✓	Multi-purpose reverse engineering tool	Reverse engineering required
		BaseSAFE [MSP20]	✓	✓	Coverage-guided fuzzing of baseband chips	Manually assembled environment
		PartEMU [Har+20]	✓	✗	Coverage-guided fuzzing of *Trust Zones*	Manually assembled environment
		Frankenstein [Rug+20]	✓	✓	Coverage-guided fuzzing of wireless firmwares	Customized for one specific device
		FIRMCORN [Gui+20]	✓	✓	Automated sandboxing of functions	Linux-based applications only
Abstraction-based	Symbolic Execution	FIE [Dav+13]	✗	✓	Symbolic execution for MSP430 microcontrollers	Complex programs lead to state explosion
		Inception [CCF18]	✗	✓	Symbolic execution, even for handwritten assembly and binary libraries	Complex programs lead to state explosion
	Concolic Execution	Concolic Testing on VP [Her+19]	✓	✓	Concolic testing of RISC-V virtual prototypes	Target must be prototyped
		Concolic Execution on Proxy [ADG20]	✓	✗	Symbolic execution on host combined with concrete execution on target	For Unix-like systems only

Overall, the wide variety of approaches in Table 3.9 demonstrates the diversity in the steadily growing research field of embedded fuzzing.

3.7 Discussion and Future Directions

Desktop user programs communicate via well-defined *syscalls* and do run in their particular *virtual address space*. Therefore, fuzzing such programs can benefit from different flavors of feedback and sanitizing options. Similarly, well-defined target constraints and boundaries are present for hardware fuzzing. Hardware fuzzing approaches can leverage hardware designs represented in Hardware Description Languages (HDLs) [Tri+22; Lae+18]. In between, embedded fuzzing faces a much less precisely specified environment. Generalized statements about interfaces, the environment, and other circumstances can not be made for embedded applications. In fact, an embedded program is an accumulation of machine code instructions that only function properly together with their intended environment and made assumptions.

This is why despite the growing attention and proliferation of embedded systems, the research field of embedded fuzzing still lacks generic solutions. Even comparing different tools remains a big challenge. It would seem that most tools are evaluated on a small set of targets, chosen by the authors themselves, whereas it would be useful to devise public, independent benchmarks.

The effectiveness of embedded fuzzers can only be evaluated when testing can be performed on a large collection of test subjects. A benchmarking suite for embedded fuzzers may consist of open-source embedded firmwares in conjunction with appropriate hardware peripheral emulation solutions. In this way, different fuzzing strategies can be evaluated on embedded systems instead of relying on the ones that are developed for user applications.

Furthermore, the different characteristics of embedded systems in contrast to user applications should be considered. Traditional fuzzing originates from quickly terminating data processing applications. Embedded systems, on the other hand, are continuously running systems that usually do not terminate after processing a single input. If the internal state of a system changes during sequences of inputs, it is called stateful. Recently, several fuzzers for stateful software have been proposed [Yu+19; PBR20; Nat22a; Sch+22c]. In particular, Pham *et al.* [PBR20] showed that stateful programs, like network servers, have to be fuzzed with awareness of their state to be efficient. Since embedded systems typically are stateful, stateful embedded fuzzing approaches are needed as well.

Most reviewed papers are emulation-based and emulators currently seem to be the preferred way of enabling embedded fuzzing. Beside their mentioned advantages, there is always the disadvantage of a lower fidelity, which makes it necessary to validate all found bugs on the actual hardware or at least an accurate model of it. This process may be automated by putting the actual device in the loop and testing input candidates directly.

The other disadvantage of emulators is the setup and configuration effort required to imitate the whole execution environment. However, with the actual hardware, there is an environment already present in which the embedded software runs as expected. Therefore, we see more research potential in performing fuzzing on the actual hardware and extracting feedback from existing functionalities e.g. debug interfaces. Common embedded debugging tools from

Lauterbach [Lau22] or *Segger* [Seg22] provide real-time tracing mechanisms for a wide variety of microcontrollers, which may be used for fuzzing feedback.

Another albeit rarely handled aspect is that an embedded system has multiple interfaces that can be highly entangled. Further research is needed to consider the entire system, and not only individual functions, interfaces, or processes while fuzzing. Such a fuzzer could fuzz on multiple interfaces simultaneously, while observing the system. Multiple fuzzers or harnesses would need to synchronize their observations, similarly to ensemble fuzzing.

Recently, plenty of automated peripheral modeling approaches, such as P^{2}IM [FML20] and Fuzzware [Sch+22b], have been proposed. For now, they seem to target rather simple embedded systems. Since they need to model all hardware peripherals that are accessed by the firmware, the approaches do not scale well for more complex systems. Nevertheless, automated peripheral modeling remains one of the most promising methods to enable generic embedded fuzzing. Further research in this area could also enable emulation-based fuzzing with low application effort for more complex embedded systems. Another option could be to design generic and reusable HALs to ease re-hosting and enable efficient fuzz testing of hardware-related code. Moreover, as highlighted by Böhme *et al.* [BCR21] for traditional fuzzing, we also advocate a larger scope for embedded fuzzers, which should identify a range of vulnerabilities, such as information and timing leakages, and not just faults.

Future research and tools should aim to unite existing techniques in an embedded ensemble fuzzing framework in order to eliminate their current, individual disadvantages. In addition, such a framework should be cross-architecture, state-aware, and compatible with emulated and real devices. Embedded Fuzzing should consider the whole system in all its details.

3.8 Related Work

Detailed summaries of the challenges of fuzzing embedded systems [Mue+18b] and security analysis of embedded systems [Fas+21; Wri+21] have been published. However, these reviews do concentrate almost solely on emulation-based approaches. We agree that emulation-based approaches are on the rise, but to get the full picture of embedded fuzzing, hardware-based approaches in all their facets need to be considered, too. We aim to draw such a complete picture and particularly want to highlight the diversity and creativity of the reviewed methods in this chapter.

3.9 Conclusion

This chapter reviewed the current state of the art of embedded fuzzing. To structure the field, we proposed a formal definition of embedded fuzzing and suggested a taxonomy for it. We carved out the additional challenges of embedded fuzzing compared to the research field of traditional fuzzing. Furthermore, we showed that no easily applicable solution for embedded fuzzing exists. As traditional fuzzing has already found numerous vulnerabilities in non-embedded software, efficient and easily applicable embedded fuzzing can increase the security and integrity of the ubiquitous embedded systems people interact with every day.

4 GDBFuzz

In this chapter, we present a new hardware-based method for fuzzing embedded systems as they are, without requiring virtualization, yet using a *unified approach* applicable to a vast variety of embedded systems. As discussed in Section 2.4, most microcontrollers contain *debug units,* through which a *debug probe* can set *breakpoints, execute* the program up to a breakpoint, and inspect the current *program state,* including the program counter and memory values. *Hardware breakpoints* are dedicated registers in the debug unit that halt the execution when the program counter equals the register value and can be set even when the *code is read only;* they neither alter nor slow down program code.

The key idea of our method is that by systematically *setting breakpoints* in the code based on the Control Flow Graph (CFG) of the program and by *checking which inputs trigger which breakpoints,* we can retrieve coverage information, and thus provide the necessary guidance for a feedback-driven fuzzing strategy. As the number of hardware breakpoints within a microcontroller is limited, we set them to a subset of the program's code blocks only, and relocate them periodically. Since many debug probes are addressable via the GNU Debugger (GDB), we have implemented the above strategy in a fuzzer named GDBFuzz, which can leverage GDB interfaces in *any* system to systematically generate test inputs guided by coverage. The intended setup is depicted in Figure 4.1.

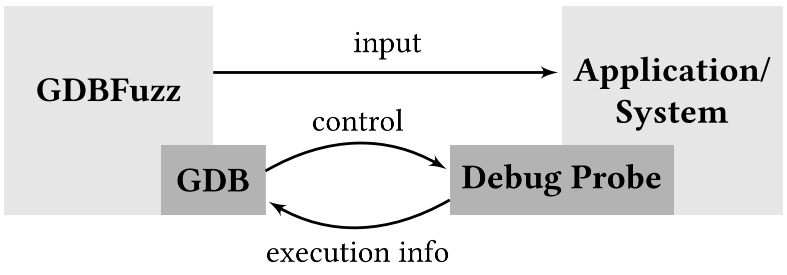


Figure 4.1: How GDBFuzz works. GDBFuzz leverages a *Debug Probe* that is connected to an *Embedded System* to control execution—notably to set (hardware) breakpoints and detect which inputs trigger which code blocks, and thus obtain coverage without having to instrument or virtualize firmware.

As it can be applied to any program and system that GDB can debug, GDBFuzz is one of the least demanding and most versatile coverage-guided fuzzers. Figure 4.2 summarizes the GDBFuzz operation. Based on the control flow graph of the target program, GDBFuzz sets the available hardware breakpoints to random nodes from the control flow graph that are yet unreached. GDBFuzz then repeatedly generates input, sends it to the target device, and checks if it triggers a breakpoint indicating new code coverage, or if it crashes the target system.

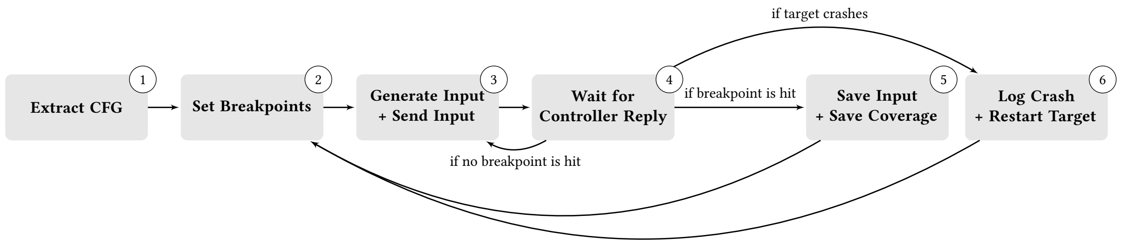

if target crashes
1 Extract CFG
2 Set Breakpoints
3 Generate Input + Send Input
4 Wait for Controller Reply
if breakpoint is hit
5 Save Input + Save Coverage
6 Log Crash + Restart Target
if no breakpoint is hit

Figure 4.2: GDBFUZZ in operation. After extracting the control flow graph (1), GDBFUZZ sets breakpoints on unreached basic blocks (2). Next, it generates a new test input, sends it to the target (3), and then waits for the execution to stop (4). If execution hits a breakpoint (= new coverage), GDBFUZZ saves input and coverage (5), sets new breakpoints (2) and keeps on fuzzing. If *no* breakpoint is hit (= *no* new coverage), GDBFUZZ tries a new input (3). If the program has crashed (6), GDBFUZZ logs it and restarts the target.

In our experiments, GDBFUZZ shows to be easily applicable on a number of microcontroller boards and even regular user applications. It achieves a much higher coverage than blackbox fuzzing and solutions based on virtualization, and also detected a number of known and new bugs. In summary, to the best of our knowledge, GDBFUZZ is the *first hardware-based, architecture-agnostic, source-code independent, non-invasive, and easy applicable method for coverage-guided fuzzing of embedded systems,* and we are happy to recommend it to anyone who wants to systematically test the robustness of embedded systems.

The remainder of this chapter is organized as follows. Section 4.1 explains control flow related mechanisms and presents the overall design of GDBFUZZ; Section 4.2 describes implementation details. We evaluate our work in Section 4.3; Sections 4.4 to 4.6 conclude and discuss the work including related and future work.

4.1 Design

Since we want to leverage hardware breakpoints for fuzzing feedback, we first need to determine at which memory addresses the target program resides. Trivially, all addresses within the executable memory regions of the device could be considered. However, only a fraction of memory addresses contain instructions that are actually executed, rendering a trivial solution ineffective for fuzzing feedback. Similar to state-of-the-art coverage-guided fuzzers, we therefore work on the *basic block level* of the target program. Hence, we extract the control flow graph from the target program, which represents *basic blocks* as nodes and describes possible transitions between them as edges. As we show in the remainder of this section, this enables us to derive *dominator relations* of control flow graphs, which help us to reduce the number of breakpoint interruptions during fuzzing and avoid unnecessary overhead.

As shown in Figure 4.2, GDBFUZZ leverages the control flow graph of the target program to set available hardware breakpoints to randomly chosen basic blocks that are yet unreached. It then repeatedly generates test cases by applying mutations to randomly drawn inputs from the input corpus, and sends the test cases to the target input interface. If the debug probe signals a breakpoint hit, GDBFuzz marks the corresponding node and its dominating nodes as reached, and adds the responsible test case to the corpus. Test cases that cause crashes or timeouts are preserved separately. When no breakpoint interrupt occurred after a predefined amount of exercised test cases, GDBFuzz relocates the hardware breakpoints to newly chosen nodes. After each relocation, GDBFuzz first tests all inputs from the corpus again to check if they already reach the newly targeted basic blocks. Like coverage-guided fuzzing with full code instrumentation, the evolutionary algorithm causes the input corpus to grow over time with inputs that reach different code areas.

The remainder of this section describes how we treat interprocedural control flow graphs and how we extract them from the target program, how we find a fuzzing entry point in the target application, and how we detect and handle bugs during execution.

4.1.1 Interprocedural Control Flow Graphs and Dominator Relations

GDBFuzz requires an interprocedural view on control flow of programs. Interprocedural control flow graphs describe additional call and return connections on top of all local control flow graphs of a program. The resulting graphs describe possible transitions of basic blocks within whole programs instead of only functions. However, constructing interprocedural control flow graphs is by far not trivial.

Simply connecting all local control flow graphs by adding all call and return instructions as edges would lead to control flow ambiguities. Considering the example program in Listing 2.2 and its local control flow graphs in Figure 2.4, for instance, and adding the two call and the two implicit return edges, would erroneously reflect control flow over the blocks 2-7-5 and 4-7-3.

More general, when a function has multiple callers a trivial construction of an interprocedural control flow graph is impossible, because it results in ambiguous paths from every calling function to every return point. Context-sensitive control flow algorithms fix that by ensuring that return edges only point back to the actual call site of the current function while traversing the control flow graph, for instance when deriving interprocedural dominator graphs [Agr99; DVD07]. However, published context-sensitive algorithms are complex and implementations are rarely available. Therefore, this thesis uses the following simple approach to express interprocedural control flow relations.

First, we construct a *semi-interprocedural* control flow graph, where we connect the local control flow graphs by inserting all calls as edges from the call site to the callee. We omit return edges when building the semi-interprocedural control flow graph to avoid introducing incorrect flow. This corresponds to the *Simple Block Model* from the reverse engineering tool *Ghidra* [Nat19]. Additionally, we compile a corresponding *reversed semi-interprocedural* control flow graph by reversing all local control flow graphs, skip call edges, and only add the reversed return edges. Again, we avoid inserting ambiguities by removing context-sensitive call edges. Figures 4.3 and 4.4 show the two control flow graphs extracted from the program in Listing 2.2.

Since the two semi-interprocedural control flow graphs reflect only valid control flow, we can derive dominator relations with the algorithms for local control flow graphs, as presented in Section 2.6. The corresponding semi-interprocedural dominator trees for the program in

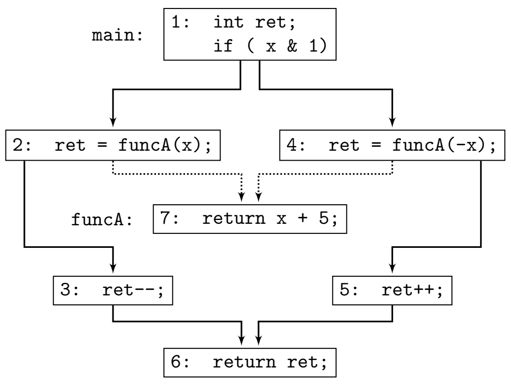


Figure 4.3: Semi-interprocedural control flow graph.

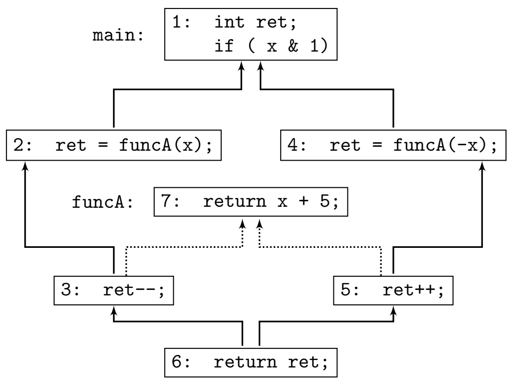


Figure 4.4: Reversed semi-interprocedural control flow graph.

Listing 2.2 are shown in Figures 4.5 and 4.6. The postdominator tree is calculated from the reversed semi-interprocedural control flow graph.

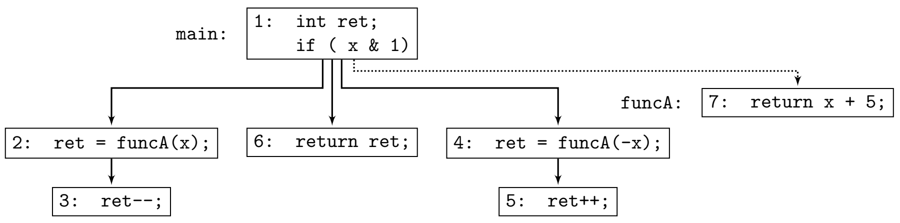


Figure 4.5: Predominator tree of the semi-interprocedural control flow graph.

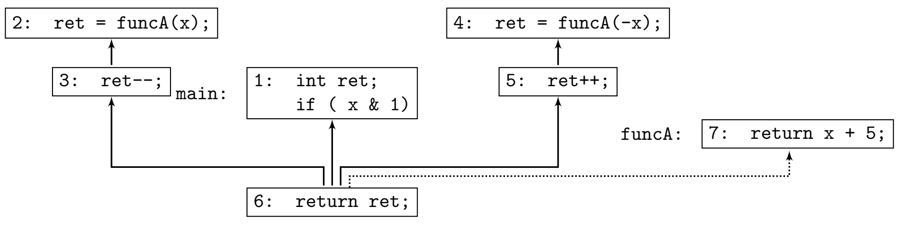


Figure 4.6: Postdominator tree of the semi-interprocedural control flow graph.

For convenience, we can merge the pre- and postdominator graph: $\{(u, v) | u \xrightarrow{pre} v \lor u \xrightarrow{post} v\}$, requiring us to only handle a single dominator graph for the whole target program. The resulting graph is not a tree anymore, but still Theorem 1 remains valid on it. Figure 4.7 shows the merged semi-interprocedural dominator graph.

Compared to context-sensitive interprocedural dominator graphs, our semi-interprocedural approach can not reflect all interprocedural relations. For instance, when a function calls two functions that call the same subroutine, our approach appends the subroutine's subtree to the parent function, although it is clear that both called functions will be executed. However, our

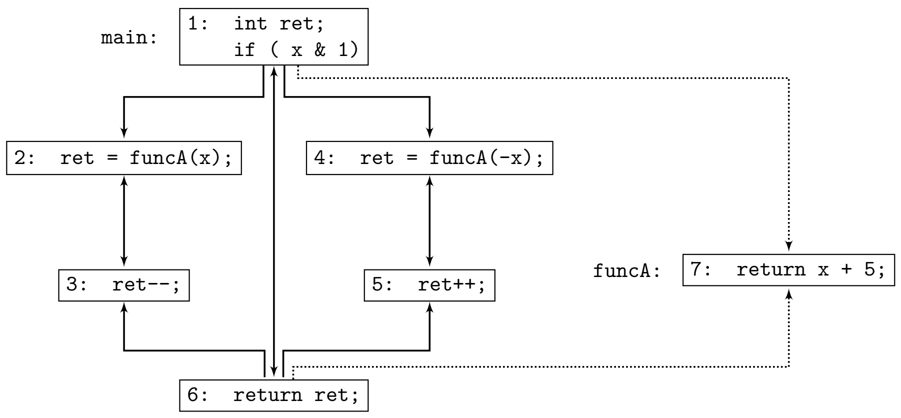


Figure 4.7: The merged pre- and postdominator trees.

approach might only miss a single of such an interprocedural dominator relation per function call in the worst case, but therefore exclusively consists of simple and comprehensive algorithms and data structures.

4.1.2 Extracting Control Flow Graphs

Control flow graphs can be obtained trivially during program compilation, because the compiler is aware of the entire control flow. However, as source code may not be available for all software components on an embedded system and to broaden its applicability, GDBFUZZ is designed to work on binaries. *Ghidra* [Nat19] is an open source reverse engineering tool which supports most common processor architectures, is scriptable, and is therefore well suited for our needs. Like all binary disassembling approaches, *Ghidra* cannot guarantee to detect all control flows, especially when it comes to indirect branches or aggressive compiler optimizations [Pan+21]. Therefore, we refine and update the control flow graph iteratively during fuzzing, which we describe in Section 4.2.

4.1.3 Finding Entry Points

We focus fuzzing on a region in the firmware where input processing of our targeted input interface occurs. Therefore, the extracted control flow graph should start at the beginning of the input processing, termed as entry point. Choosing the entry point is a task for the test engineer, who thus requires knowledge about the target. However, the following semi-automated way of finding a suitable entry point has turned out to be useful in our analysis.

1. Send a test input to the target device and interrupt the execution immediately.
2. Use `gdb find` to rediscover the test input in the memory of the device.
3. Set a data watchpoint to the first address of the rediscovered input data.
4. Send the test input again.

5. All program counter addresses on now occurring interrupts are candidates for an entry point.

These steps are conducted once, as part of the GDBFuzz setup.

GDBFuzz can also work with *symbol names* as entry point if they are included in the binary, to avoid the need of searching the entry point after each re-compilation. This is particularly useful in continuous integration setups, such that new software versions can be fuzzed seamlessly.

4.1.4 Detecting Bugs

Bug oracles detect whether a bug is triggered during execution. Since fuzzing origins from testing user applications, a common bug oracle is to observe the target process on raised error signals, e.g. segmentation faults. To find bugs that do not trigger faults directly, sanitizers and assertions are used. These are usually deployed at compile time, but there are methods to inject sanitizers directly into binaries [Din+20]. However, more sophisticated bug oracles are still an open research problem [BCR21] and out of scope of this work.

GDBFuzz relies on the triggered bugs being observable, meaning that faults or other misbehavior must be triggered by the bug. Silent corruptions, as demonstrated in [Mue+18b], can therefore not be discovered, unless additional sanitizers are used during compilation. Faults can be detected, for instance, by occurring connection errors like timeouts or error response codes. Additionally, breakpoints can be set on locations of fault handlers via the debugging interface, which also catches fault signals from deployed sanitizers.

Since the location of the fault handlers in the code commonly does not change during runtime, software breakpoints can be used to detect their execution such that all hardware breakpoints are available for the coverage feedback mechanism. Software breakpoints are well suited in this case, since they are not repositioned during fuzzing. If software breakpoints are not available for the SUT, a subset of available hardware breakpoints can be used, too, with the disadvantage of decreased fuzzing performance[1].

Obtaining the locations of fault handlers is done in a manual to semi-automated way, because embedded systems vary dramatically in features, such as processors, operating systems, frameworks, libraries, and sanitizers. In our experience, the default fault handlers e.g. from FreeRTOS [Fre22], Arduino [Ard22a], and STM32CubeMX [STM22] typically end in an infinite loop. *Ghidra* can identify functions with infinite loops [Nat22b], which can then be considered as potential fault handlers. For all our test applications, it was sufficient to rely on timeouts that are provoked by the infinite loops in the fault handlers, not requiring any further setup work for us.

4.1.5 Handling Bugs

Whenever GDBFuzz detects a crash or a timeout, it

1. *deduplicates* the bug to identify whether the bug is *unique*, i.e. whether it is the first time that this bug was found;

[1]We evaluate the influence of the number of available breakpoints on performance in Section 4.3.

2. *preserves* the input triggering this bug if the bug is unique;
3. *restarts* the target system; and
4. *continues* fuzzing.

The same bug may be triggered multiple times during fuzzing. Analyzing each bug requires substantial efforts, which is why deduplication is required. The goal is to provide the test engineer a minimal set of inputs triggering only unique bugs. We use hashes of the call stack [Man+19] to uniquely identify and deduplicate bugs.

When a bug is triggered, the target system may be in a non-recoverable state. Similarly, if a timeout occurs, the target system may hang forever. For this reason, we reset the target system via GDB after a fault has been discovered.

4.2 Implementation

GDBFUZZ consists of the following components:

Test Data Generator. Like all coverage-guided fuzzers, GDBFUZZ preserves inputs that trigger different code areas in the input corpus, and derives new inputs by mutating these. Dozens of general purpose mutation-based fuzzers have been published in recent years [Man+19]. We therefore do not develop a mutation algorithm from scratch, but reuse the mutation engine from libFuzzer [LLV15]. The actual mutation engine in GDBFUZZ is easily interchangeable.

GDB Controller. The *GDB controller* manages the debugging connection to the SUT. Common debug probes usually provide a *GDB Server* via a TCP socket. We use the Python package *python-gdb-mi* for sending and receiving debugging commands, like setting breakpoints or continuing the execution.

Target Connection. The *target connection* component is an abstraction for sending test inputs to the target device. It handles connection or disconnection events depending on the actual interface. It also handles error feedback from the protocol. Embedded systems can feature tons of different input interfaces and channels from where untrusted input is consumed. Popular interfaces include *Wi-Fi*, *Bluetooth*, *NFC*, but also all kind of external facing buses like *CAN*, *USB*, *Profibus*, or I^2C. GDBFUZZ can include custom interface adapters to enable a broad applicability. For our case study, we implemented adapters for *TCP*, *Serial*, and *USB* connections, as well as *UNIX pipelines* to enable fuzzing of Linux applications.

Ghidra Controller. We use the reverse engineering tool *Ghidra* to obtain the control flow graph of the target application. For interchanging requests and data between *Ghidra* and GDBFUZZ, we use the *ghidra-bridge* Python package. GDBFUZZ can connect to a running *Ghidra* instance or start a headless instance on the target binary.

Dynamic Control Flow Graph Refinement. As mentioned earlier, reverse engineering tools cannot guarantee to detect the whole control flow of a program [Pan+21]. Missing control flow manifests itself as a dangling node in the control flow graph without successor that is not marked as *terminal* by *Ghidra*. When finding a test input that triggers the execution of such a dangling node, we perform the following steps:

1. Set a breakpoint to the dangling node and send the test input that triggers it to the SUT.
2. When the interrupt occurs, perform a single step.
3. Read the value of the program counter.
4. Report the found edge to *Ghidra* and reanalyze the binary.

Ghidra is then usually able to recover even more control flow based on the reported edge.

4.3 Evaluation

In this section we evaluate GDBFuzz in two different settings, guided by eight research questions (RQs).

1. For the hardware-based setting we pick a variety of common development boards, listed in Table 4.8 together with their corresponding architectures, utilized debug probes, and the number of available hardware breakpoints.

Table 4.8: Details of our development boards including architecture, utilized debug probe, and the number of available hardware breakpoints.

Board	Architecture	Debug Probe	#HW Br.
STM32L4S5I [STM20a]	ARM	STLinkv3	6
CY8CKIT-062-WIFI-BT [Inf18]	ARM	KitProg3	6
ESP32-DevKitC_V4 [Esp16]	Xtensa	J-Link Ultra	2
EXP430F5529LP [Tex13]	MSP430	eZ-FET lite	8

On each development board we deploy four different classes of applications, listed in Table 4.9 with an initially given seed. The *Buggy* program exposes the buggy function from Listing 2.1 to a serial input interface and serves as a ground truth. The specific applications for each board are derived from examples shipped with the development boards, or compatible tool chains[3]. The *HTTP* and *USB* application classes require the appropriate interface to exist on the target board.

2. The application-based setting features 16 programs from Google's Fuzzer Test Suite [Goo16a], compiled as *x86* Linux applications, and provides a scalable and independently measurable

[2]Empty USB Command Block Wrapper (CBW) frame

[3]The actual applications and corresponding references are in the replication package

Table 4.9: Application classes and initial input seeds for our case study on embedded hardware.

Name	Description	Seed
Buggy	Buggy program from Listing 2.1	None
JSON	Parses serial data as *json* string	"1000, 2000, 3000"
USB	USB mass storage client	`55 53 42 42 00`...[2]
HTTP	HTTP server via Wi-Fi	"GET / HTTP1.1"

evaluation environment. GDBFuzz can execute an application either in a QEMU instance, enabling live measuring of reached code coverage, or with GDB directly, enabling low overhead and unlimited amounts of breakpoints[4]. We compile the applications with compiler optimizations (`-O3`), and execute the corresponding experiments on a server with four Intel Xeon Gold 6144 CPU's and 1.48 TB of RAM.

4.3.1 GDBFuzz vs. Blackbox Fuzzing

GDBFuzz enables coverage-guided fuzzing on systems where coverage measurement is hardly possible[5]. We therefore utilize the partial coverage extraction mechanism of GDBFuzz itself, to measure coverage differences between GDBFuzz and blackbox fuzzing in our hardware-based setting. Specifically, we deploy GDBFuzz, but omit adding new inputs to the input corpus during fuzzing to simulate a blackbox fuzzer. As a result, we can investigate how the evolutionary fuzzing algorithm of GDBFuzz performs, and can consequently address our first research question:

RQ1: How does GDBFuzz compare against blackbox fuzzing on embedded systems?

Figure 4.10 shows coverage over time plots for all board and application class combinations. Each experiment is repeated twice, leading to an accumulated experiment time of 56 days. Without exception, GDBFuzz achieves a higher code coverage across all runs than blackbox fuzzing and shows that it can greatly benefit from the partial coverage information it retrieves via hardware breakpoints. In particular for the *Buggy* program, blackbox fuzzing has little to no chance to fulfill all conditions to trigger the contained stack overflow bug, as theoretically described in Section 2.2. In this application class, GDBFuzz achieves almost 100 iterations per second on the powerful *CY8CKIT* board, while it can only reach about 1.5 iterations per second on the low performance *MSP430* board. This explains why it takes way longer for GDBFuzz to solve the input constraints on the latter, and we can also see how important throughput is for fuzzing. Nevertheless, GDBFuzz performs well on all of our development boards, finds the bug in all cases, and reports the resulting crashes properly.

[4]QEMU theoretically enables an unlimited amount of breakpoints, too, but suffers from an increasing execution overhead.

[5]Otherwise we would use the available mechanism for coverage-guided fuzzing

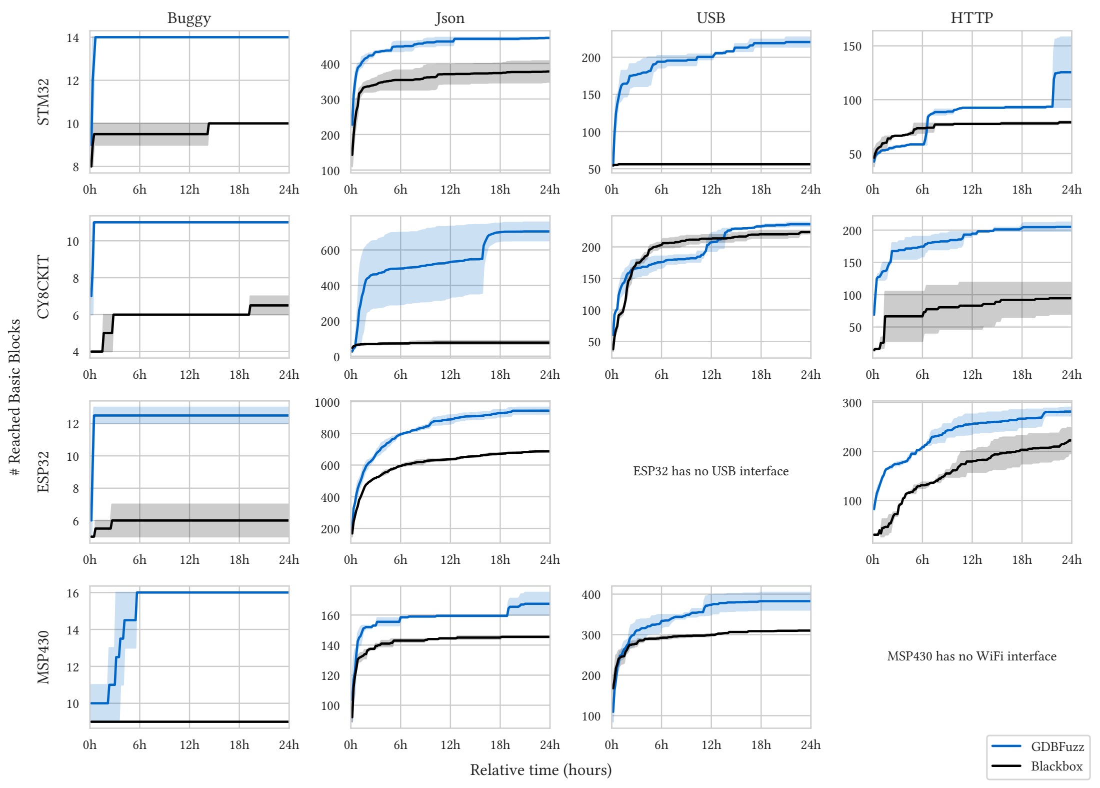


Figure 4.10: Reached basic blocks over time for GDBFuzz and blackbox fuzzing on embedded hardware (N=2).

Takeaway 1

Coverage-guided fuzzing with a limited amount of breakpoints is effective and outperforms blackbox fuzzing on embedded systems.

4.3.2 GDBFuzz vs. State of the Art

As showed in Chapter 3, there are multiple approaches that claim to enable coverage-guided fuzzing for embedded systems, raising the following question:

RQ2: How does GDBFuzz compare to existing embedded fuzzing methods?

μAFL [Li+22] is a hardware-based embedded fuzzer that uses the ARM Embedded Trace Macrocell (ETM) interface to extract code coverage from an embedded program. Our *CY8CKIT-062-WIFI-BT* development board features such an ETM interface, and we have access to the tracing hardware required therefore. However, the publicly available version of μAFL reported implausible results on our setup, which we could not resolve despite having vendor support. We noted that μAFL uses raw trace data functions, which are unreliable in some implementations, and are marked for *internal use only* [Gmb19]. Also, and in contrast to GDBFuzz, μAFL requires very specific hardware, which is why it is not a direct competitor; we are not aware of a generic hardware-based embedded fuzzing approach to compare GDBFuzz against.

Most of published embedded fuzzing methods are emulation-based, from which only *peripheral modeling* approaches can enable coverage-guided fuzzing for embedded systems on a scale and are competitors to GDBFuzz. We therefore compare GDBFuzz against the latest peripheral modeling approach Fuzzware [Sch+22b], whose authors claim embedded fuzzing on the actual hardware to be impractical. Fuzzware works on all *ARM Cortex-M*-based microcontrollers, so we can fuzz all applications from the first two development boards in Table 4.8.

First, we need to agree on how we compare emulation-based to hardware-based approaches. Li *et al.* [Li+22] compared *peripheral modeling* approaches to their hardware-based approach μAFL by the number of the achieved fuzzing iterations per hour. We agree that the number of executions per time is an important metric for fuzzing. However, for a fair comparison, the same or at least similar code areas must be executed in that time. Peripheral modeling approaches like P^{2}IM [FML20] and Fuzzware [Sch+22b] use fuzzing to iteratively carve an artificial execution environment for the firmware. Over time the peripheral models are refined, and the execution speed decreases since the firmware can be further executed. By design, peripheral modeling does not target specific code areas. This makes throughput a meaningless measure for comparing these different approaches, because it is unclear whether the same code parts are executed in this time.

We therefore compare embedded fuzzing approaches based on the number of reached basic blocks in a targeted region of the firmware during fuzzing, as also done in [Sch+22b].

Emulation-based approaches can be scaled up easily by using multiple cores, which is more complex and expensive with hardware-based approaches. To let Fuzzware benefit from its scalability, we assign 16 cores for each trial, while GDBFuzz runs as a single instance for the same amount of time. Afterwards, we evaluate how many basic blocks from the target regions have been reached by Fuzzware and GDBFuzz.

Table 4.11 lists the absolute and relative number of reached basic blocks using Fuzzware and GDBFuzz. Although it had significantly more computing power provided, Fuzzware did not reach *any* basic block on six out of eight applications, whereas GDBFuzz covered a substantial part of them. On the remaining two applications, GDBFuzz reached more basic blocks than Fuzzware. The USB controller on the STM32 board transfers data via DMA, which is not supported by Fuzzware, but is required to execute the application. From our experience DMA is a widely used mechanism to interact with hardware peripherals, and the lack of DMA support by Fuzzware is a major drawback. The *Wi-Fi* and Transmission Control Protocol (TCP) protocol handling on the STM32 board takes place in a separate chip connected via Serial Peripheral Interface (SPI) to the microcontroller. In order to trigger the execution of the HTTP parser, Fuzzware would need to model the inter chip communication protocol correctly, which it did not.

On the *CY8CKIT* board Fuzzware can not execute any application, because the boot phase of the *CY8CKIT* development board requires interaction between the two contained processors, which Fuzzware is not able to model.

Table 4.11: Covered basic blocks by Fuzzware with 16 cores, and GDBFuzz after 24 hours of fuzzing.

	Target	Basic Blocks Covered			
		Fuzzware		GDBFuzz	
STM32	buggy	11/17	(64.7%)	**14**/17	**(82.3%)**
	json	435/560	(77.7%)	**472**/560	**(84.3%)**
	usb	0/518	(0%)	**220**/518	**(42.5%)**
	http	0/166	(0%)	**126**/166	**(75.9%)**
CY8CKIT	buggy	0/13	(0%)	**11**/13	**(84.6%)**
	json	0/1217	(0%)	**704**/1217	**(57.8%)**
	usb	0/456	(0%)	**236**/456	**(51.8%)**
	http	0/402	(0%)	**205**/402	**(51.0%)**

For complex embedded programs with DMA and complex boot routines, more computing power will not lead Fuzzware to reach the targeted code. In general, the more complex the targeted input interface, the harder it is for peripheral modeling approaches to provide reasonable fuzz data.

Takeaway 2

GDBFuzz fuzzes software on embedded systems without requiring any instrumentation or other software change.

4.3.3 Reveal Bugs with GDBFuzz

The main goal of fuzzing is to find software bugs, which leads to the question:

RQ3: Can the method reveal actual bugs in embedded software code?

For answering **RQ3**, we first have a look at two known real-world bugs. The USB enumeration handling of the STM32CubeL4 USB Middleware [STM20b] contains the known vulnerabilities *CVE-2021-34259* and *CVE-2021-34268* that were found using μAFL [Li+22]. We verify that GDB-Fuzz can detect such real-world vulnerabilities and use a firmware based on the USB host Mass Storage Device Class (MSC) example application version 1.17.1 from STM32CubeL4 [STM21]. The STM32 microcontroller acts as USB host in this MSC application. Our setup therefore is similar to that from the μAFL authors. To generate USB traffic, we plug a common USB flash drive as USB client into the USB port. We then introduce a fuzzing harness into specific stages of the USB enumeration, where we replace the USB data frame with fuzz data just before the USB host processes this data. Namely, we replace the raw *device descriptor* or the *device configuration* that is sent by the client device before any parsing of the fuzz data. The introduced fuzzing harness receives fuzz data from GDBFuzz via a serial interface. Both mentioned *CVEs* manifest themselves as timeout when triggered, because the USB host middleware gets wrongly configured by malformed *device descriptor* or *device configuration* USB packets. Basically they are caused by missing validity checks for the untrusted data from the USB client. With GDBFuzz and the appropriate fuzzing harnesses, both *CVEs* are triggered and detected in less than 5 minutes during our experiments.

During evaluation, we discovered three previously unknown bugs, which we reported to the corresponding vendor:

1. An infinite loop in the STM32 USB device stack, caused by counting an `uint8_t` index variable to an attacker controllable `uint32_t` variable within a *for loop* [Eis22c].
2. A buffer overflow in the Cypress JSON parser, caused by missing length checks on a fixed size internal buffer [Eis22a].
3. A null pointer dereference in the Cypress JSON parser, caused by missing validation checks [Eis22d].

Takeaway 3

GDBFuzz reveals real vulnerabilities in embedded software.

4.3.4 GDBFuzz vs. AFL++

The application-based setting allows us to fuzz Linux applications with GDBFuzz, which raises the research question:

RQ4: How does GDBFuzz compare against the state-of-the-art fuzzer AFL++?

For a fair comparison between GDBFuzz and AFL++ [Fio+20], we let them operate on the uninstrumented binary using QEMU mode for AFL++ (`-Q`) and GDBFuzz with QEMU, too. As live measurement is impossible with the modified QEMU version included in AFL++, we replay the respective input corpora after fuzzing and measure the reached number of basic blocks thereby. We configure GDBFuzz to use eight breakpoints, which is a realistically low number of breakpoints available in real microcontrollers.

Figure 4.12 shows coverage over time plots for GDBFuzz and AFL++. Obviously, AFL++ covers more code over time than GDBFuzz. AFL++ is designed and optimized for exactly this kind of applications and can benefit from its exhaustive code instrumentation. However, we think that GDBFuzz with just eight utilized breakpoints is not too far away. On some

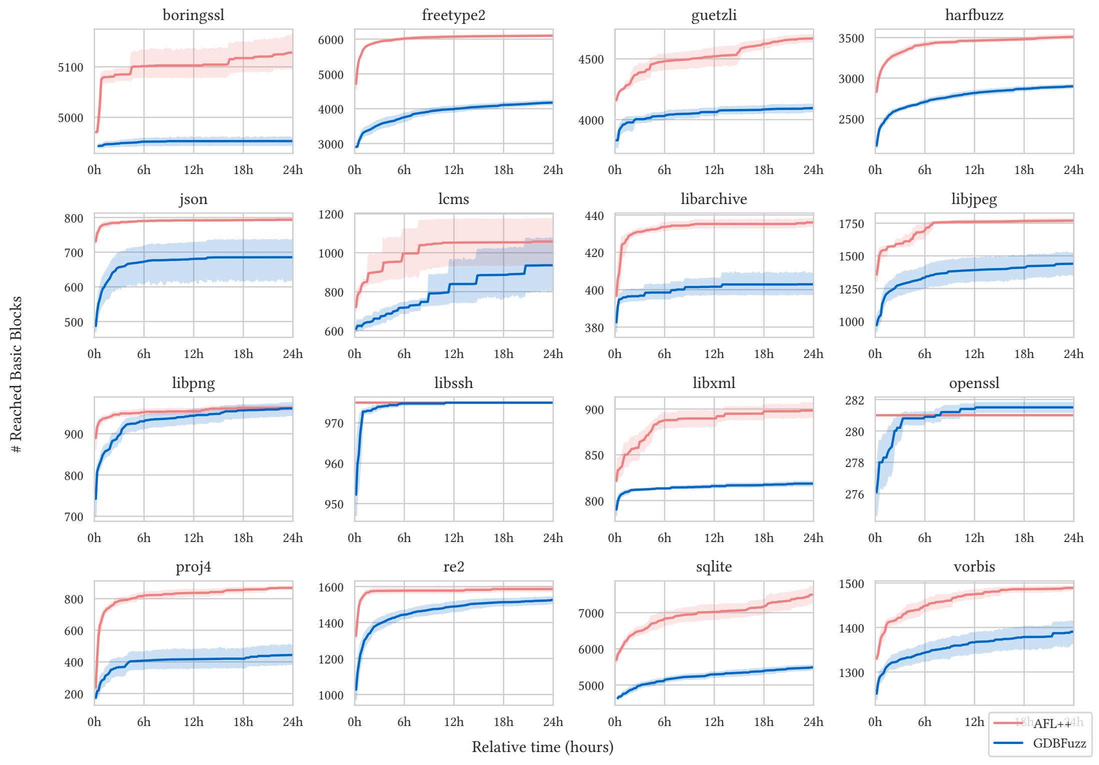


Figure 4.12: Basic block covered by GDBFuzz and AFL++ on applications where code instrumentation is possible (N = 10).

application, GDBFuzz could even reach a similar number of basic blocks. We also emphasize that AFL++ falls back to *blackbox fuzzing* in scenarios where emulation and instrumentation is not available—and this is again where GDBFuzz is superior.

Takeaway 4

If one can deploy AFL at little cost, use it; otherwise, consider GDBFuzz as a potentially less demanding alternative.

4.3.5 Boost by Dominator Relations

Let us now evaluate specific elements of the GDBFuzz design.

RQ5: How much does GDBFuzz benefit from dominator relations?

To answer **RQ5**, we analyze the average number of breakpoint interrupts, as well as the average number of reached basic blocks during the previous experiments in Table 4.13.

Table 4.13: Averaged results of the benchmark (N=10).

Target	#Interrupts	Basic Blocks	Precision	New Blocks	New Edges
boringssl	511.6	1398.8	99.69%	+674.16%	+708.61%
freetype2	1281.1	3537.8	99.91%	+433.22%	+460.42%
guetzli	258.3	2274.4	99.68%	+21.52%	+21.36%
harfbuzz	1086.2	2668.2	99.88%	+13.86%	+14.11%
json	306.5	736.0	97.69%	+0.0%	+0.0%
lcms	229.6	813.3	99.04%	+12.24%	+12.2%
libarchive	152.3	431.0	99.26%	+0.36%	+0.23%
libjpeg	534.0	1403.7	98.96%	+148.02%	+150.58%
libpng	370.4	1050.5	97.21%	+0.03%	+0.02%
libssh	303.2	1013.5	99.61%	+19.29%	+19.94%
libxml	260.8	732.1	98.73%	+0.05%	+0.21%
openssl	109.8	286.7	100.0%	+0.64%	+0.6%
proj4	182.6	437.8	99.8%	+3.26%	+3.15%
re2	589.1	1613.0	99.88%	+4.13%	+3.95%
sqlite	1080.0	4364.3	98.19%	+1.59%	+1.65%
vorbis	400.44	1387.0	99.57%	+12.77%	+12.36%

Across all our experiments, each breakpoint interrupt led to 3.15 marked basic blocks on average, meaning that the number of probed basic blocks is reduced by 68.25%. This ratio is better than in experiments of the efficient code instrumentation algorithm presented in [TH02], where the authors achieved to reduce the number of instrumentation points only by 34% to 49% in their experiments. GDBFuzz can presumably reduce the overhead further, because we additionally use post dominator relations and the described semi-interprocedural control flow graph.

As we can see in the *Precision* column, the vast majority of the marking dominating basic blocks was correct. Incorrectly marked basic blocks can result from incorrect reverse engineered control flow. The reached precision of mostly more than 99% is sufficient for coverage-guided fuzzing since it is a stochastic process and does not rely on 100% correct coverage data. Popular fuzzing tools, like AFL++, store coverage data in hash maps and also miss some coverage during fuzzing due to hash collisions.

Takeaway 5

Across over our experiments, dominator relations reduced required breakpoint interruptions by more than two thirds.

4.3.6 Revealing Control Flow

GDBFuzz can guide reverse engineering tools to reveal undetected control flow, as we described in Section 4.2, and we investigate by the question:

RQ6: How well can GDBFuzz aid reverse engineering tools to reveal unrecognized control flows?

We answer **RQ6** by comparing the average relative number of additional revealed basic blocks and edges against the ones that *Ghidra* initially detects. Since the targets have been compiled with activated compiler optimizations, recovering the control flow is particularly hard for reverse engineering tools. In Table 4.13, we can see that up to 674.16% additional basic blocks and 708.61% additional edges could be revealed during our experiments. A lower number of newly found control flow does not necessarily show a lower performance from GDBFuzz, but rather a good reverse engineering performance of *Ghidra*.

Takeaway 6

GDBFuzz reveals undetected basic blocks and edges for reverse engineering during fuzzing.

4.3.7 Impact from the Number of Available Breakpoints

The amount of available breakpoints varies across different microcontroller families and models, which raises the question:

RQ7: How does the number of available breakpoints affect the fuzzing performance?

For estimating how different numbers of breakpoints influence the fuzzing performance, we execute the applications directly with GDB and use ordinary software breakpoints for the feedback, since QEMU does not scale well with an increasing amount of breakpoints. This way we have arbitrarily many breakpoints available without impacting execution time, and can estimate how their number influences the achieved coverage over time. To answer **RQ7**, we execute GDBFuzz in the application-based setting using an exponentially increasing number of virtual breakpoints from 1 to 65536. Representative, Figure 4.14 shows the reached basic blocks over time on four applications[6], averaged from 2 runs.

[6]Plots for all other applications available in the repository

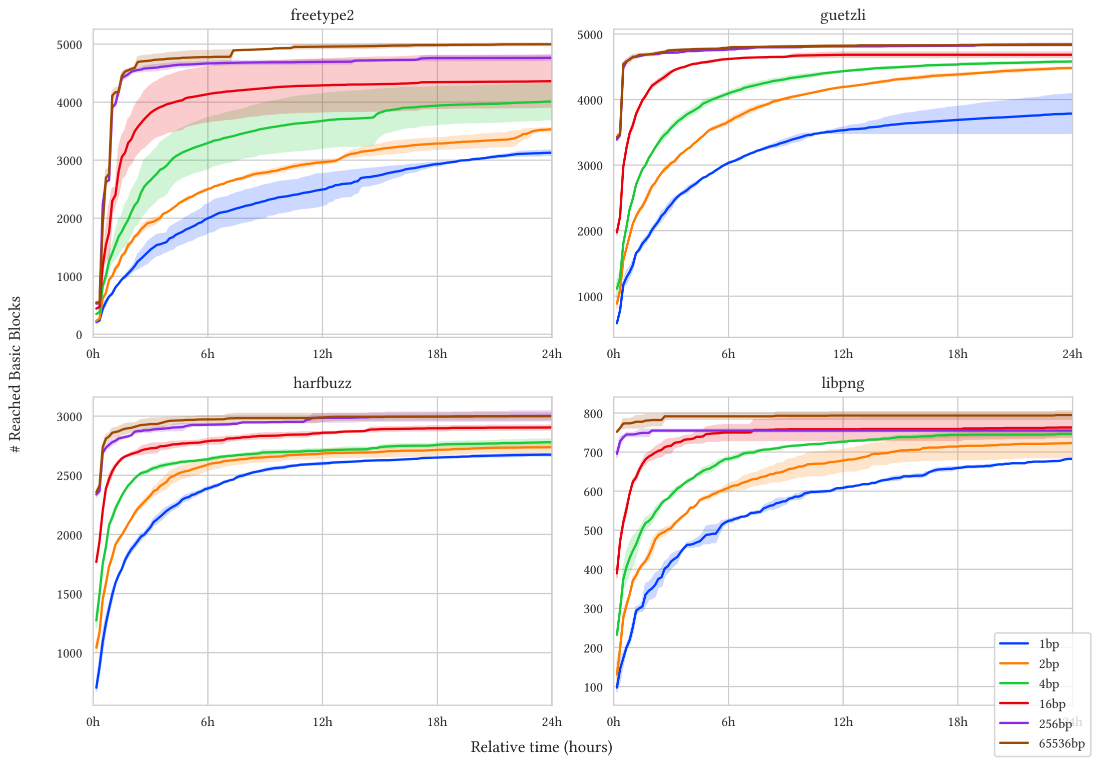


Figure 4.14: Fuzzing performance of GDBFuzz on four applications using different numbers of virtual breakpoints (N=2).

Unsurprisingly, more used breakpoints lead to more covered code blocks per time. In our experiments, it roughly seems that doubling the number of breakpoints yields to a linear improvement of fuzzing performance. Exponential correlations between effort and revenue are common in the research area of fuzzing [BF20]. Likewise, our experimental observation between the number of utilized breakpoints and coverage over time suggests an exponential correlation.

Takeaway 7

Linearly more coverage over time requires exponentially more breakpoints.

4.3.8 GDBFuzz vs. Blackbox Fuzzing

As pointed out in Section 4.3.4, AFL++ falls back to *blackbox fuzzing* when no conforming instrumentation mechanism is available. This motivates our final research question:

RQ8: How does GDBFuzz compare to blackbox fuzzing on the application-based setting?

To compare GDBFuzz against blackbox fuzzing, we measure reached basic blocks directly during fuzzing using QEMU, because there is no corpus to replay for blackbox fuzzing.

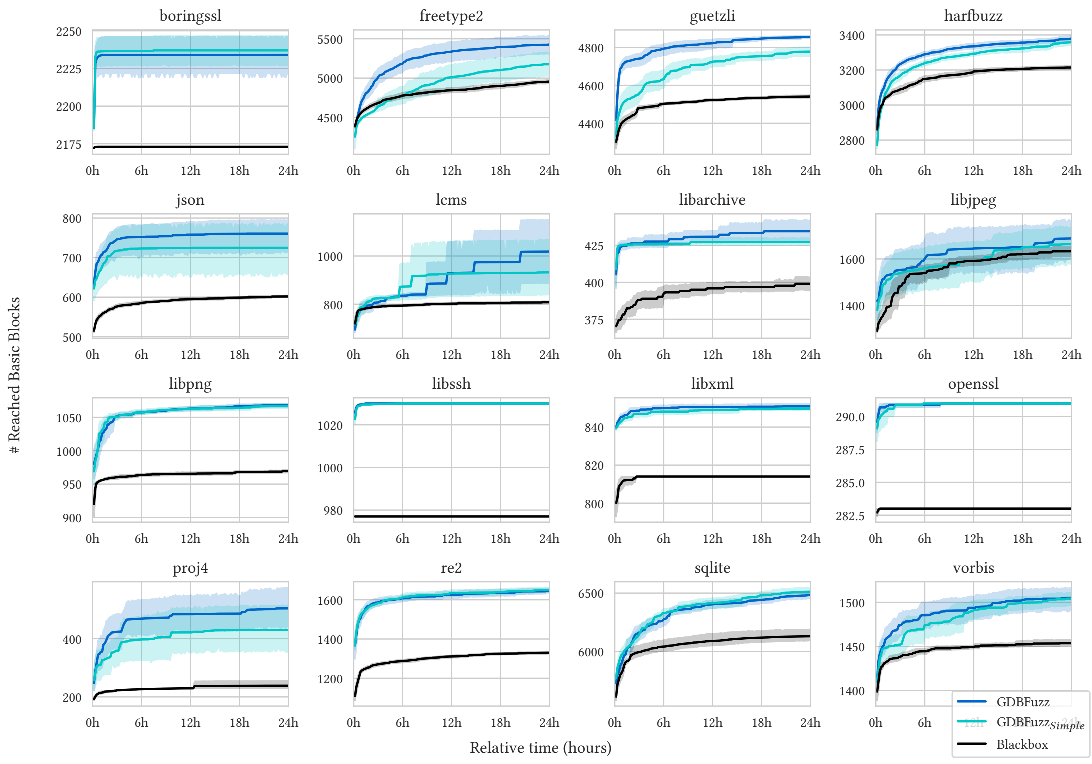


Figure 4.15: Reached coverage from GDBFuzz with and without using dominator relations, and blackbox fuzzing measured by QEMU (N = 10).

Figure 4.15 shows measured coverage over time results for blackbox fuzzing, GDBFuzz, and a trimmed version (GDBFuzz$_{\text{Simple}}$) that does not make use of dominator relations as described in Section 2.5. This larger scale benchmark with independent code coverage measurements confirms the results from the development boards: GDBFuzz outperforms blackbox fuzzing in all experiments. Furthermore, we can see that using dominator relations to gain transitive knowledge led to more and faster code coverage on most target applications. Nevertheless, even GDBFuzz$_{\text{Simple}}$ greatly outperforms blackbox fuzzing in all experiments.

Takeaway 8

GDBFuzz outperforms blackbox fuzzing in an application-based setting on a larger scale, too.

4.4 Discussion

Now that we showed how well GDBFUZZ works, we want to discuss the design choice of using code block coverage as metric, how fuzzing with GDBFUZZ works in practice, and the threats to validity of our evaluation experiments.

4.4.1 Block and Branch Coverage

While most state-of-the-art fuzzers like AFL++ [Fio+20] and libFuzzer [LLV15] leverage edge coverage, we use block coverage to guide the evolutionary fuzzing algorithm. Extracting edge coverage with our methods, would require to probe already reached blocks multiple times, which would increase overhead drastically. Nagy *et al.* [NH19] use software breakpoints to detect the execution of new basic blocks for normal software. They find that edge coverage cannot benefit from its finer granularity because of the required constant instrumentation overhead. Since edge instrumentation would introduce even more overhead in our setting than source code instrumentation on normal software, we estimate block coverage as the only feasible coverage metric in GDBFUZZ.

4.4.2 Fuzzing Firmware Drivers

Fuzzing drivers, or the middleware of embedded software, can be implemented in a blackbox approach, by injecting fuzz data to external facing interfaces, or as a whitebox approach, by compiling a fuzz harness into the firmware to fetch and redirect the fuzz data to the driver function. Whitebox approaches offer more flexibility since driver functions can be called directly and an acknowledgment signal can be fed back to the fuzzer, which indicates a function has properly returned. However, expert knowledge about the code and the system is required to implement a suitable fuzzing harness. Also, an implemented harness works on a distinct code base only, and false positives can be produced since input can be sanitized in the hardware already before the tested driver function is reached [Li+22]. A whitebox approach was suitable for replicating the known *CVE's*, revealed by μAFL, since the faulty functions were known and the original finders have chosen the same way.

A blackbox approach usually requires less setup effort, because data routes to the targeted interface should exist in most test setups anyway, and the test engineer therefore just needs to connect an existing route to GDBFUZZ by implementing a suitable Python class. As a consequence, our connection adapters work out of the box e.g. for all existing USB device drivers. The previously unknown bugs we found with GDBFUZZ, have been triggered without the development and use of extensive harnessing functions, but on the unchanged firmware.

4.4.3 Threats to Validity

Empirical studies are necessarily fraught with threats to validity. To address *external validity* doubts, we have tested the method on different development boards with different debuggers and architectures and made sure that we can find real-world bugs. Additionally, we conducted a larger case study on known fuzzer benchmarking targets. To minimize the risk of systematic errors and addressing *internal validity* doubts, we decoupled coverage measurements from our tool during evaluation and repeated each experiment multiple times. The implementation of GDBFuzz is publicly available to allow reproduction of our results.

4.5 Related Work

There are approaches that use *software breakpoints* for measuring code coverage and obtaining fuzzing feedback, in order to avoid the overhead and impediments of source code instrumentation [Nag+21; NH19; Gro20]. The idea is to insert software breakpoints into unreached basic blocks and therefore allow the program to execute at full speed until new coverage is reached. Once the execution runs into a breakpoint, the corresponding instruction is removed from the binary to avoid further overhead. Oh *et al.* [Oh+15] use *software breakpoints* to measure code coverage in embedded firmware. They extract the start address of each basic block of the program during compilation and insert software breakpoints at each of them, also removing them once they are hit. The advantage of using software breakpoints is that their number is unlimited, but in contrast to hardware breakpoints they require rewriting of the firmware image on each change.

4.6 Conclusion and Future Work

The field of embedded fuzzing lacks generic, easy applicable, and efficient solutions. We propose a debugger-driven fuzzing method that relies only on the presence of a *GDB* compatible debug probe and hardware breakpoints on the microcontroller. GDBFuzz therefore enables cheap, non-intrusive, and source code agnostic coverage-guided fuzzing on embedded systems. It is designed to work out of the box for a wide variety of microcontrollers and input interfaces. In contrast to earlier assertions, we showed that hardware-based embedded fuzzing is practical and reveals software bugs. As fuzzing is performed on the raw hardware, execution is fast and naturally accurate. Detected failures are real and can be easily replicated.

We evaluated our implementation GDBFuzz on four embedded application classes featuring four different microcontrollers, showing that it beats blackbox fuzzing and the latest emulation-based approach Fuzzware in all cases. Furthermore, we tested GDBFuzz in an emulated environment on popular fuzzer benchmarking targets to gain more experiment data and statistics. We showed that leveraging dominator relations boosts the performance of GDBFuzz, and that already a single hardware breakpoint is sufficient for enabling coverage-guided fuzzing. We also showed that GDBFuzz can reveal control flow that is missed by a reverse engineering tool during fuzzing. All in all, if an embedded system provides a debugger interface, GDBFuzz provides a practical fuzzing solution.

Future work on GDBFuzz could focus on enhanced strategies for choosing basic blocks to probe and incorporating established fuzzing optimizations, like *corpus minimization, dictionaries,* or different *seed schedules.* Also, fuzzing for stateful embedded systems could be considered in future work.

5 GDBMiner

In this chapter, we will focus on model-driven fuzzing, where the main challenge is to create proper input models. We will use context-free grammars as input models, since they are simple to define, but can represent rich properties of an input specification. Recent research has demonstrated that input grammars can be *mined* from inputs and programs:

- **Autogram** [HZ17] traces data flow of different parts of the input into functions and variables and compiles resulting rules to a context-free grammar. However, it is implemented for Java programs only, and therefore, unsuited for a variety of systems and programs.
- **Mimid** [GMZ20] leverages control flow, as well as data flow instrumentation of the target program to reconstruct derivation trees, and subsequently uses a number of active learning steps—basically testing for interchangeability of subtrees—to translate these into a grammar. Mimid requires the program code to be written in Python or C (Cmimid). Cmimid requires the C code to:

  1. Not use macros and enumerations;
  2. Have individually separated case statements with braces and without fall-throughs; and
  3. Avoid GOTO statements.

  While such restrictions pose little problems for the proof-of-concept of a prototype, they make Cmimid impractical for industrial use. Even if the source code is available, it cannot be easily rewritten to match Cmimid requirements.
- **Arvada** [KLS21] is a *black-box* approach that only requires oracle requests (tests whether an input is accepted by the target program) to approximate a grammar. It therefore offers an approach to grammar mining that only requires the ability to execute a program. However, by construction, Arvada has less information available than code-based approaches, and the resulting grammars may thus be less precise.

To mine input grammars in our context, we have created a novel approach that does not suffer from the above limitations—and actually is set to be applicable for binaries and executables in *any* programming language, on *any* operating system, using *any* processor architecture, even without source code. The key ingredient of our approach is, again, to use the *GNU Debugger (GDB)* as interface to the program under test, allowing a grammar miner to:

1. *Step* through the program, identifying the code executed.
2. Use *watchpoints* to *track* data accesses during execution.

These features give our grammar miner the ability to *associate* input data with *code locations* that process them. This, in turn, allows a grammar miner to *group* input bytes that are processed by the same code into *equivalence classes* and hence *grammar elements.* Also taking the call stacks of the processing code into account, we can produce precise and human-readable input grammars.

Using GDB to step through executions is time-consuming—but once a grammar is mined from a system, it can be used again and again for high-speed generation of valid inputs, offsetting any effort spent to mine the grammar in the first place. As explained in Section 2.4, GDB supports *remote debugging*, where GDB runs on one machine, and the program under test runs on another. GDB then communicates with a remote *GDB Server* on the program under test that understands the GDB protocol through a serial device or TCP/IP. This is particularly useful for embedded devices, as the stub and the communication require only few resources. Furthermore, it allows our approach to even mine grammars from programs on embedded systems.

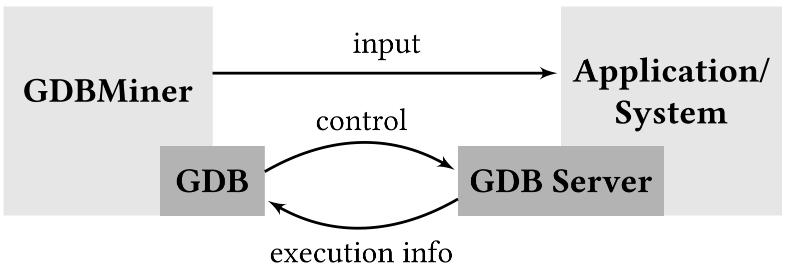


Figure 5.1: GDBMiner connects to a *GDB*-compliant *GDBServer* in order to set data watchpoints and trace the execution of *inputs* within an *Application* or *System.*

We have implemented the above grammar mining approach in an open source tool named *GDBMiner,* depicted in Figure 5.1, bringing input grammar mining to any program that can be debugged with GDB—from C-compiled executables on general-purpose PCs to read-only binaries on embedded systems.

With GDBMiner, we contribute a *unified, language- and architecture-agnostic approach* for mining input grammars on *any* system with debugging capabilities. As we show in our evaluation, GDBMiner produces input grammars that are precise, well-structured, and very suitable for test generation, especially for black-box testing. The versatility of GDBMiner and the universality of the resulting grammars enables efficient software testing under adverse conditions, including embedded systems; and thus specifically addresses the needs of software engineering in practice.

The remainder of this chapter is organized as follows. Section 5.1 presents the idea of our approach. We cover implementation details in Section 5.2. Section 5.3 evaluates GDBMiner against the state of the art. Section 5.4 closes with conclusion and future work.

5.1 Design

Mining input grammars refers to deriving grammars from programs that match their input space best possible. Mimid [GMZ20] and Arvada [KLS21] are currently, to the best of our knowledge, the two most effective methods to do so. Both require a set of valid seed inputs as a starting point. Seeds for a program are obtained by collecting example inputs that are available or by intercepting messages to the program during runtime.

Mimid [GMZ20] applies comprehensive instrumentation in terms of control and data flow instrumentation to the target code to find at which point of the execution the program *consumes* which bytes of the input data. It determines the consumption of a character using dynamic tainting to even track when the program accesses eventual copies of input bytes or extracted tokens of a lexer stage. For the control flow instrumentation, Mimid introduces a stack that, similar to a call stack, keeps track of active control flow scopes, such as *loops*, *if/else branches*, and *function calls*. From tracking the execution while parsing a set of valid seed inputs and in conjunction with the documented input data accesses, Mimid recovers derivation trees, which already remind on parse trees of grammars. Mimid then searches for compatible nodes in these trees by swapping their subtrees and probing if the resulting new input is accepted by the target program. Mimid's source code aware approach enables the extraction of meaningful and human-readable labels from symbols, but it requires exhaustive instrumentation.

In contrast, Arvada [KLS21] is a *black-box* approach and contents itself with oracle requests whether an input is accepted by the target program or not. Also, based on a set of valid seed inputs, it tries to condense symbols in the input to *bubbles*, meaning they are assigned a new non-terminal symbol, and subsequently tries to *merge* different bubbles when they turn out to be compatible. Arvada chooses candidates for new bubbles randomly, which makes it a non-deterministic approach with respect to the seed inputs.

Similar to the Mimid algorithm, we exploit that the call stack of a recursive descent parser should express a branch of the derivation tree when a symbol is *consumed*; that is when the program processes a character of the input buffer last. The call stack in this case does not only contain the called functions but additionally all taken control flow decisions like conditional branches and loops. Cmimid determines the consumption of a character using LLVM's data flow sanitizer [LLV22] that can even track when the program accesses eventual copies of input bytes. In contrast to Mimid, we relax the condition to consider a symbol to be consumed and just take the point where the program accesses the character in the input buffer last. This allows us to use data watchpoints for tracking accesses to the input buffer and therefore a programming language independent and source code agnostic approach. Moreover, it allows us to run the method on arbitrary systems that offer standard debugging capabilities.

Figure 5.2 shows the different stages of GDBMiner for mining grammars. In short, we use the *single-step* feature of the debugger to trace the execution of the targeted program and additionally log accesses to the input buffer with watchpoints. Based on the tracing data, we first reconstruct all control flow graphs of the involved functions, detect loops and conditional branches, and reconstruct derivation trees for each input. If the system under test offers only a limited amount of hardware watchpoints, we trace the same input multiple times with a sliding

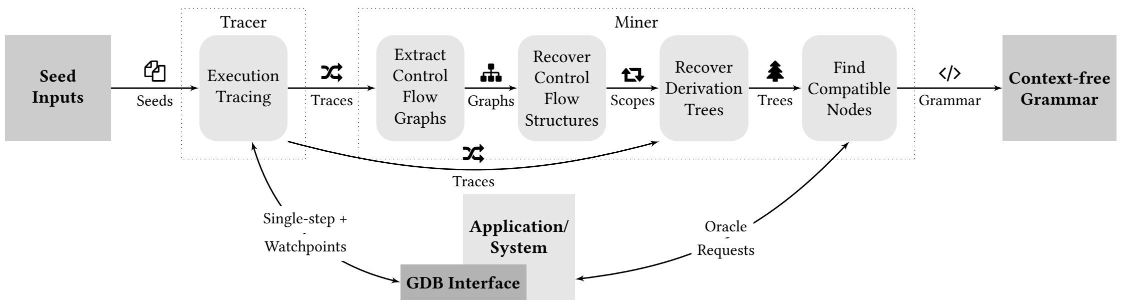


Figure 5.2: How artifacts emerge through the different stages of GDBMiner. By tracing the execution of the *Seeds*, GDBMiner obtains *Traces*, from which it derives *Control Flow Graphs*. From the graphs, GDBMiner recovers *Control Flow Scopes*, which are required to reconstruct the *Derivation Trees*. Finally, GDBMiner extracts a *Grammar* through active learning.

window of available watchpoints. Finally, we take the recovered derivation trees and apply the Mimid mining algorithm, consisting of multiple generalization steps.

5.1.1 Tracing

Before we can trace the parser of the target program, we need to determine the location of the input buffer in memory that the program reads from. The location of the input buffer depends on the concrete program and the execution environment, which is why we require the user to specify a symbol name or the actual address. To trace only the parsing stage rather than tracing the entire program, it is advisable to provide an entry point and optionally an exit point. Good candidates are parser functions, which usually have "*parse*" in the name and take a character buffer pointer as a parameter, which in turn holds the input buffer location. We explain more details in Sections 5.2.1 and 5.3.4.

Having the address of the input buffer and an entry function, we can start tracing the processing of the input. First, we let the program execute until the entry function using a breakpoint. On reaching the entry point, we byte-wise assign watchpoints on all addresses of the input buffer. For the actual tracing, we use the single-step functionality of GDB to walk through the program under test instruction by instruction. At each interrupt, we document the current program counter address and the current stack trace. In addition, we document eventual watchpoint hits (accesses to the input buffer) and reference them to the latest trace entry. We repeat this procedure until we step out of the entry function or we reach a defined exit point. Consequently, each trace element consists of a program address, a list of return addresses on the stack, and a list of watchpoints hits. To obtain a complete set of traces, we execute these steps for each seed input once.

5.1.2 Recovering Control Flow Graphs

Since we need to detect loops and conditional branches in our target program, we first need to recover the Control Flow Graphs of all involved functions. The Control Flow Graph expresses possible control flow sequences of a function as a directed graph from the entry to the exit node. Statically recovering Control Flow Graphs from binary programs is its own discipline in academia [Pan+21]. However, since we have single-step traces available, we can dynamically recover the relevant parts of the Control Flow Graph from a binary program.

We therefore iterate over the list of executed instructions and stack traces. If the stack length remains equal between two subsequent instructions, we add the transition as edge to our graph. When we observe an increase of the stack length, the last instruction must have been a function call. We ignore the call edge and connect the call instruction with the return address that is saved on the stack. When we observe a decrease of the stack length, we do not add anything to the graph. This way the Control Flow Graph of a function stays consistent and, most importantly, within the function itself.

5.1.3 Recognizing Control Flow Structures

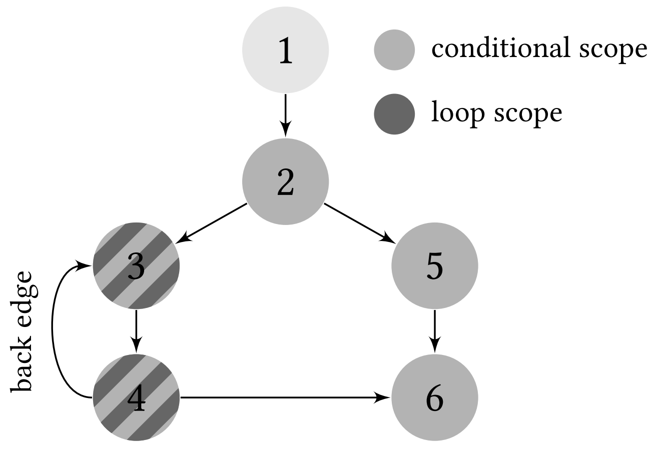


Figure 5.3: Conditional and loop scopes in a simple control flow graph.

Based on a Control Flow Graph $G = (V, E)$, and the predominator relation from Definition 8, we can locate back edges in G [ASU07]:

Definition 12: *[Back Edge]*

An edge $(u, v) \in E$ is a *back edge*, if the head v predominates its tail u, namely $v \xrightarrow{pre} u$.

Given a back edge (u, v), the corresponding natural loop is the set of nodes that can reach the tail u without going through the head v [ASU07], or rather the nodes that can still reach u, when we remove v from the graph:

Definition 13: *[Natural Loop]*

The *natural loop* of back edge (u, v) consists of the nodes:
$\{w \mid w \in V \text{ and } \mathit{reachable}\,(G \setminus \{v\}, w, u)\}$.

Having these efficient computable properties, we can find all loops in Control Flow Graph G by iterating over all edges, checking whether it is a back edge, and, if that is the case, associate the corresponding natural loop with the head of the back edge.

A node with multiple successors in G is a conditional branch. We define the scope of a conditional branch as:

Definition 14: *[Conditional Scope]*

The *conditional scope* of node u, where $|G[u]| \geq 2$ is: $\bigcup_{v \in G[u]} \{w \mid v \xrightarrow{pre} w\}$.

The scope of a conditional node therefore is the union of all dominated nodes of its successors. Figure 5.3 shows loop and conditional scopes in a relatively simple control flow graph with a single conditional branch and a single loop. We can easily identify the back edge that closes the loop between nodes 3 and 4.

5.1.4 Recovering Derivation Trees

Based on the single-step traces, the Control Flow Graphs of all functions, the detected natural loops, and the scopes of conditional branches, we can now start to recover a *derivation tree* for each seed input. We want to represent the execution flow of a parser in these derivation trees, particularly which function stack is responsible for which input character. Similar to grammar parse trees, derivation trees therefore have only non-terminal symbols as leaf nodes. As Mimid showed, we can recover derivation trees by analyzing the control flow of the parsing program. Since we cannot benefit from the extensive instrumentation used by Mimid, we need the additional processing step, described in Algorithm 2, to translate single-step traces into a derivation tree.

First, we initialize the list that represents the current call stack (Line 1), as well as the dictionary that represents the derivation tree we want to build from the trace (Line 2). Starting from Line 8 we process each trace element iteratively. For each element, we first check if the program address belongs to the start of a function, and if so, append that function and its scope to the stack and the currently active tree branch. Next, in Line 12, we test whether the current address does not belong to the uppermost scope on the stack. If that is the case, we remove all left scopes from the stack. Similar to entered functions, we check if the current address enters a new loop scope (Line 15) or a conditional scope (Line 19) and also append the respective scope addresses to the stack and the tree. If the current trace element contains watchpoint hits, we add the corresponding index as a leaf node to the current active tree branch in Line 22. After exercising the whole trace, the tree might contain some input buffer indices multiple times and

Algorithm 2: Recover derivation trees from traces.

```
   Input: A single-step trace
   Output: A derivation tree to the trace
1  STACK ⟵ [(0, [])]                                   # init stack
2  TREE ⟵ {}                                          # init tree
3
   # helper function to add a scope to the tree
4  def add_scope(id, scope):
5      TREE[STACK[-1].id].append(id)                   # add to parent
6      STACK.append ((id , scope))                     # push to stack
7
   # iterate over trace addresses
8  foreach id, addr ∈ trace do
9      if addr ∈ func_entries then                    # function start
10         add_scope(id , function_scope(addr))
11     while addr ∉ STACK[-1].scope do                # check scope
12         STACK.pop()                                # drop exited scope
13     if addr ∈ loop_entries then                    # loop start
14         add_scope(id , loop_scope(addr))
15     S ⟵ CFG[addr]                   # get successors of addr from CFG
16     if |S| ≥ 2 ∧ S ⊆ STACK[-1].scope then
           # start of a conditional scope
17         add_scope(id , cond_scope(addr))
       # attach eventual watchpoint hits
18     foreach idx ∈ watchpoint_hits(id) do
19         TREE[STACK[-1].id].append(idx)
```

branches that do not have such indices at all. We keep only the last access to each byte of the input buffer and discard others. In a separate step, we remove all branches in the tree that do not end with an input buffer index, resulting in our desired derivation tree.

Let us consider the JSON array parser in Listing 5.1 as an example for typical parser code. The function `parse_val` consecutively tries to match the character at the current cursor position within a large switch statement. When the value starts with a square bracket, it calls the `parse_array` function, if nothing matches it parses the current value as a number. The function `parse_array` first checks if the character at the current cursor ends the array with a closing squared bracket and returns if that is the case. If not, it repeatedly calls `parse_val` as long as values are followed by a comma. If it reads a closing squared bracket, it ends the loop and returns.

Figure 5.4 shows the derivation tree that our approach recovers with the explained algorithm from the JSON parser code processing the string *'[1,2,3]'*. The inner nodes of the tree represent

Listing 5.1: JSON parser (array parser excerpt). Adopted from [Sch17].

```

int parse_val(char **cursor ) {
  ...
  switch (**cursor) {
  ...
  case '[': {
    ++(*cursor);
    valid = parse_array(cursor);
    break;
  }
  default: {
    double number = strtod(*cursor);
    ...
  }
  }
}
int parse_array(char** cursor) {
  ...
  int valid = true;
  if (**cursor == ']') {
    ++(*cursor);
    return ;
  }
  while (valid) {
    valid = parse_val(cursor);
    if (!valid) break;
    ...
    if (has_char(cursor, ']')) break;
    else if (has_char(cursor, ',')) continue;
    else valid = false;
  }
  ...
}
```

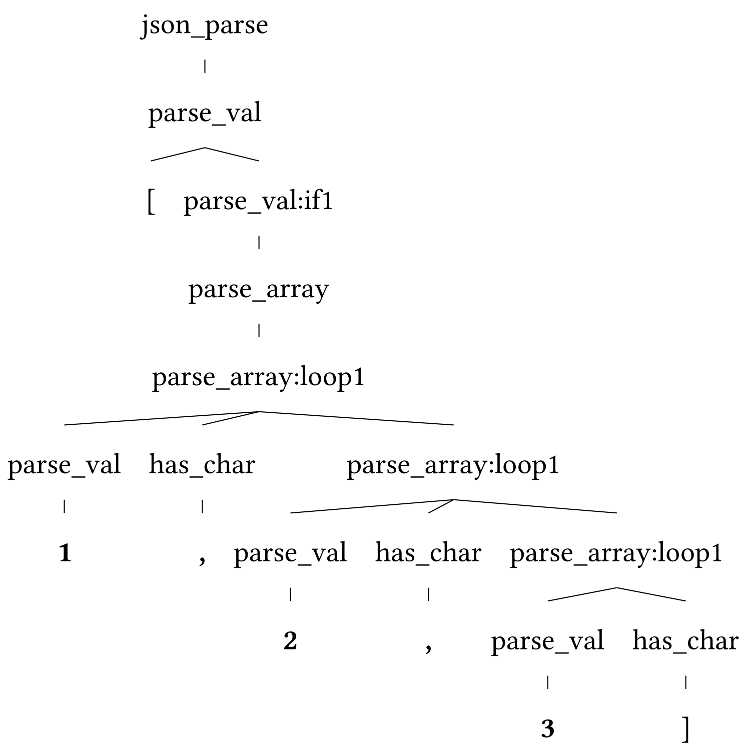


Figure 5.4: Recovered derivation tree from *JSON* with input *'[1,2,3]'* using Algorithm 2.

functions or control flow structures of functions. Since we cannot simply distinguish between switch statements and if/else constructs in a trace, all conditional branches are named as "*if*" e.g. the case statement of function *parse_val* is identified as *parse_val:if1*. Similarly, we name label all flavors of loops, e.g. *while*, *do while*, and *for* loops, as "*loop*". Mimid puts each new iteration of a loop on the same level in the tree. Our approach differs from Mimid in treating multiple loop iterations. Notably, we can see that the program processes digits followed by a comma in the loop *parse_array:loop1* and that our approach attaches subsequent loop iterations as a child branch of the previous one. From our observation, loops in parsers are usually not limited by a constant value, but consume input until a certain character is reached. For instance, in the case above, the *parse_array:loop1* loop ends reading the character *']'*. Nesting the loop iterations leads to more generalizing grammars, as we explain in the next section. The resulting tree expresses at what point the program uses which characters of the input buffer.

5.1.5 From Derivation Trees to a Grammar

Given a derivation tree, we can derive a context-free grammar by treating all leaf nodes as terminals, all inner nodes as non-terminals, and create production rules for each inner node to the concatenation of all its child nodes. We can merge multiple grammars by union the sets of variables, terminals, and productions. The resulting grammar matches the input language of the program as long as all inner nodes of the derivation tree with the same name are compatible with each other. Hence, interchanging them still results in accepted inputs. In practice, however, this is rarely the case.

Like Mimid, we use an *active learning* stage, where we examine exactly this compatibility among identically named nodes. For each pair of identically named nodes (a, b), we:

1. Replace the subtree of node b with the one of node a.
2. Extract the resulting input string.
3. Check if the target program can parse the input.

Additionally, we cross-check if node a is replaceable by node b. If in both cases the program accepts the input, we consider the nodes compatible and assign them a common name, if not, we assign different names. Since we process all possible pairs of nodes, the worst case complexity of this approach is quadratic.

Applying this process to the derivation tree in Figure 5.4, we figure out that all *parse_val* nodes are interchangeable. In contrast, the first *has_char* node is not compatible with the last *has_char* node, and we need to distinguish them in the resulting grammar.

$$
\begin{aligned}
\langle \mathit{START} \rangle &::= \langle \mathit{json\_parse} \rangle \\
\langle \mathit{json\_parse} \rangle &::= \langle \mathit{parse\_val} \rangle \\
\langle \mathit{parse\_val} \rangle &::= \text{‘\texttt{[}’}\langle \mathit{parse\_val{:}if1} \rangle \mid \text{‘\texttt{1}’} \mid \text{‘\texttt{2}’} \mid \text{‘\texttt{3}’} \\
\langle \mathit{parse\_val{:}if1} \rangle &::= \langle \mathit{parse\_array} \rangle \\
\langle \mathit{parse\_array} \rangle &::= \langle \mathit{parse\_array{:}loop1} \rangle \\
\langle \mathit{parse\_array{:}loop1} \rangle &::= \langle \mathit{parse\_val} \rangle\langle \mathit{has\_char1} \rangle\langle \mathit{parse\_array{:}loop1} \rangle \mid \langle \mathit{parse\_val} \rangle\langle \mathit{has\_char2} \rangle \\
\langle \mathit{has\_char1} \rangle &::= \text{‘\texttt{,}’} \\
\langle \mathit{has\_char2} \rangle &::= \text{‘\texttt{]}’}
\end{aligned}
$$

Figure 5.5: The grammar derived from the tree in Figure 5.4.

Figure 5.5 shows the preliminary grammar derived from the derivation tree in Figure 5.4. We can see that the *has_char* node appears as ⟨*has_char1*⟩ and ⟨*has_char2*⟩ in the resulting grammar, depending on whether another loop iteration ⟨*parse_array:loop1*⟩ follows. The grammar can already serve for generating valid *JSON* arrays of arbitrary length, including nested arrays and the digits *1*, *2*, and *3*.

Usually, we have multiple seed inputs and therefore multiple derivation trees to learn a grammar from. For a correct grammar, we have to apply the aforementioned active learning steps on all equally named nodes across all obtained derivation trees. Then, we simply merge the individual grammars into a single one.

However, the mined grammar is still cluttered and by far not minimized. To clean the grammar, we remove non-terminals with single rules and replace them with their children accordingly. Additionally, we use the Mimid algorithm to generalize terminal symbols by replacing them with predefined dictionaries of symbols (digits, letters, punctuation) and verifying that the program still accepts generated inputs. Finally, we check if we can replace a symbol with multiple characters and insert appropriate replication rules.

5.2 Implementation

Our implementation consists of the *Tracer* component that takes care of generating traces for different systems and the *Miner* component that exercises the recovering of the control flow graphs, the control flow structures, the derivation trees, as well as performing active learning steps adopted from Mimid.

5.2.1 Tracer

We use the Python *pygdbmi* package to control the execution of the target system or program via GDB debug commands. Common instruction set architectures support one to sixteen hardware watchpoints [Gre+12], forcing us to trace each seed input repeatedly while sliding a window of available watchpoints over the input bytes. For Linux user programs, Valgrind [NS07] offers the usage of an unlimited amount of virtual watchpoints. It does so by letting the target program run in an emulator leading to precise control over all its memory accesses. This allows us to cover each byte of the input buffer individually with a watchpoint to recognize accesses during tracing on Linux. Another option would be, e.g., QEMU's user mode emulation [Bel05].

To save time, we only single-step through functions we are interested in. Therefore, we start tracing at a manually defined entry function and stop tracing when we step out of the entry function or at a manually defined exit point. Additionally, we offer to blacklist functions that get skipped during tracing. When a called function f matches the blacklist regular expression, the tracer skips f including any subroutines using the GDB *finish* command, which continues the execution until f exits (step out). Frequent blacklist candidates, for instance, are functions like `malloc` or `free`. Defining the location of the input buffer in form of a symbol name or memory address also is a manual task. While this sounds like tedious work, the proceeding therefore is quite simple when using the *find* function of GDB to locate input bytes in memory:

1. Start the program with GDB, set a breakpoint to the `main` function, and continue the execution.

2. Step through the program until the *find* function of GDB finds the input we execute the program with. Note the address of the found input buffer.

3. Set a watchpoint to the first address of the input buffer.

4. Restart the program, wait for a watchpoint hit, and take the first address of the current function or the function on the stack as entry point.

5. Take the address of the last instruction of the chosen entry function as exit point.

Alternatively, a suitable entry function and the corresponding input buffer can be determined from reading function signatures when there is source code available.

The entry point, the blacklisted functions, as well as the input buffer location are defined by their symbol names or alternatively by their actual addresses. The latter is of interest when tracing a binary program without debug symbols.

A concrete trace consists of a list of trace entries that each logs the current *address*, *function name*, *function arguments*, *stack*, and eventual *watchpoint hits*.

Since we rely on single stepping for generating the traces, we require the watchpoint implementation to trigger interrupts even during that operation mode, which unfortunately is not true for the ARMv7 debug unit [ARM21]. We therefore have to investigate the state of the watchpoint registers after every single-step. On ARMv7 processors the corresponding bit is in the `DWT_FUNCTION` register, which we can simply read with the GDB *examine* command. Therefore, after every single step, we examine the corresponding bits of the debug unit and attach eventually triggered watchpoint events to the current trace entry.

5.2.2 Miner

The miner component starts with a set of raw traces and first recovers the control flow graphs of functions in the target program, as explained in Section 5.1.2. Next, it extracts all *natural loops*, as explained in Section 5.1.3. With these prerequisites, we now exercise Algorithm 2 to obtain a derivation tree for each of the traces.

As detailed in Section 5.1.5, the next task for the miner component is to discover compatible subtrees from the set of derivation trees. Therefore, we require the program to reflect whether the input was valid or not during or after execution. On Linux programs, we simply consider the input to be valid if the *exit code* is zero, which is the common convention. For embedded programs, however, we require a protocol-specific feedback mechanism. Error codes or behavior is common for most protocols. However, we simply stick to a serial connection that responds with zero when the input is valid or a negative number if parsing fails for our evaluation setup. This feedback oracle suffices to exercise active learning steps, as explained in Section 5.1.5.

5.3 Evaluation

For evaluating GDBMiner, we first consider the eleven programs listed in Table 5.6 as a case study for our evaluation, along with handwritten *golden grammars* that cover the program's input specifications. We obtained these from the published replication packages from Mimid [GMZ20] and the referenced open source repositories. The programs thereby use language-specific control flow structures, such as `setjmp` and `longjmp` in C, `std::exception` and `std::visit` in C++, and Rust's `match` and `std::result` mechanisms.

Table 5.6: Programs for our case study.

Program	Accepted input	PL	Origin
From Cmimid replication package			
cgidecode	CGI-style escaped strings.	C	[GMZ20]
json	JSON format.	C	[GMZ20]
mjs	Micro Java Script programs.	C	[GMZ20]
tinyc	TinyC programs.	C	[GMZ20]
Our additional selection of programs			
calc	Arithmetic operations.	C	[Bui10]
xml	XML format.	C	[Hel13]
calcrs	Arithmetic operations.	Rust	Original
jsonrs	JSON format.	Rust	[Lin16]
calccpp	Arithmetic operations.	C++	[Fah17]
jsoncpp	JSON format.	C++	[Oku09]
xmlcpp	XML format.	C++	[Kal06]

The informed reader surely recognizes that XML is *not* a context-free format because it requires matching opening and closing tags of arbitrary length.[1] However, we adhere to the subset of XML from [KLS21] for our evaluation, which limits possible tags to the characters *a, b, c, d.* In practice, programs usually consume inputs with such a limited set of possible tags, making them context-free and suitable for our approach.

We compile all programs in debug mode (`-O0`) to keep most of the structure from the source code, all symbol names, and most importantly a consistent stack at all times. We examine stack frames during tracing to determine function calls and return addresses, which get messed up when using compiler optimizations. Using reverse engineering tools like Ghidra [Nat19] would allow GDBMiner to operate on optimized or even obfuscated binaries, but that is out of scope for this work.

As in [GMZ20; KLS21] we measure the quality of the mined grammars as *precision* and *recall* values. We therefore generate 1,000 inputs from the mined grammars and test how many generated inputs the target program accepts. The resulting *precision* value therefore is the number of accepted inputs divided by the number of tested inputs. Subsequently, we generate 1,000 inputs from the golden grammar and test how many of them are parsable by the mined

[1]Can be shown with the Pumping Lemma.

grammars, using the Early parsing algorithm [Ear70]. The corresponding *recall* value is the number of parsable inputs divided by the number of generated inputs.

Achieving a high precision score alone is easy, because one can trivially construct a grammar that just generates the seed inputs leading to 100% precision. On the other hand, simply getting a high recall value is easy as well, because a grammar that can generate any string gets a recall value of 100%. The harmonic mean between precision and recall values, known as F_1*-score*, condenses the two performance values into a single accuracy value we use to compare the different approaches.

5.3.1 Resulting Grammars

First, we have a look at grammars that GDBMiner creates. Figure 5.7 shows a grammar

⟨*START*⟩ ::= ⟨*parse_sum.0-1*⟩

⟨*parse_sum.0-1*⟩ ::= ⟨*parse_mult.0-1*⟩ | ⟨*parse_mult.0-1*⟩ ⟨*parse_sum.1-0-c*⟩ ⟨*parse_sum.0-1*⟩

⟨*parse_mult.0-1*⟩ ::= ⟨*parse_primary.0-c*⟩ | ⟨*parse_primary.0-c*⟩ ⟨*parse_mult.0-0-c*⟩ ⟨*parse_mult.0-1*⟩

⟨*parse_sum.1-0-c*⟩ ::= ‘+’|‘-’

⟨*parse_primary.0-c*⟩ ::= ‘(’⟨*parse_sum.1-1*⟩ | ⟨*DIGIT_s*⟩

⟨*parse_mult.0-0-c*⟩ ::= ‘*’|‘/’

⟨*parse_sum.1-1*⟩ ::= ⟨*parse_mult.0-1*⟩ ⟨*parse_sum.1*⟩

⟨*parse_sum.1*⟩ ::= ‘)’ | ⟨*parse_sum.1-0-c*⟩ ⟨*parse_sum.1-1*⟩

⟨*DIGIT_s*⟩ ::= ⟨*DIGIT*⟩ | ⟨*DIGIT*⟩ ⟨*DIGIT_s*⟩

⟨*DIGIT*⟩ ::= ‘0’|‘1’|‘2’|‘3’|‘4’|‘5’|‘6’|‘7’|‘8’|‘9’

Figure 5.7: Mined grammar for valid math expressions.

mined by GDBMiner from the *calc* program using 20 seed inputs. We can see that GDBMiner recovered the recursive nature of mathematical expressions and that it can generate arbitrary deep expressions, e.g. with the sequence of non-terminals ⟨*START*⟩ ⟶ ⟨*parse_sum.0-1*⟩ ⟶ ⟨*parse_mult.0-1*⟩ ⟶ ⟨*parse_primary.0-c*⟩ ⟶ ⟨*parse_sum.1-1*⟩ ⟶ ⟨*parse_mult.0-1*⟩. With ten non-terminals and 26 production rules, the mined grammar comes close to the handwritten grammar, which requires six non-terminals and 21 production rules. More importantly, the resulting grammar is correct and has reasonable names for non-terminals.

Table 5.8: Precision values in percentage averaged from 30 runs.

Program	Cmimid	Arvada	GDBMiner
cgidecode	**100**±0	**100**±0	**100**±0
json	**100**±0	82.89±6.81	**100**±0
mjs	**98.01**±3.77	87.59±11.19	91.67±6.47
tinyc	**100**±0	78.07±15.02	67.15±9.28
calc	**100**±0	99.99±0.06	**100**±0
xml	N/A	31.13±14.46	**98.59**±0.52
calcrs	N/A	**100**±0	**100**±0
jsonrs	N/A	76.53±9.03	**97.20**±3.75
calccpp	N/A	**100**±0	99.86±0.75
jsoncpp	N/A	78.48±8.40	**100**±0.
xmlcpp	N/A	50.41±23.90	**96.41**±1.80

Takeaway 9

Grammars from GDBMiner are human-readable.

5.3.2 Comparison against State of the Art

Next, we want to compare GDBMiner against the current state-of-the-art grammar miners Mimid [GMZ20] and Arvada [KLS21]. We therefore compile the programs as *Linux user applications* that read input from *stdin* and return a non-zero value if the parsing fails. Our test hardware for this part of the evaluation is a server with four Intel Xeon Gold 6144 CPUs and 1.48 TB of RAM.

As in [KLS21] we randomly generate 20 seed inputs from the golden grammar and let the three grammar mining approaches operate on exactly the same set of seed inputs. We repeat each experiment 30 times to compensate for the probabilistic nature of seed generation and the approaches itself, average the achieved precision and recall values, and calculate the standard deviation.

Table 5.8 shows the achieved precision values. Cmimid achieves the highest precision on all the five programs it can compile. Deriving input grammars from control flow is very precise, but Cmimid has high requirements that the *xml* C source code does not fulfil. Cmimid does not support C++ and Rust at all.

Table 5.9: Recall values in percentage averaged from 30 runs.

Program	Cmimid	Arvada	GDBMiner
cgidecode	**100**±0	93.62±3.05	**100**±0
json	52.08±7.41	**66.32**±12.34	61.70±10.32
mjs	57.96±31.56	59.38±40.98	**82.66**±27.56
tinyc	50.95±9.95	**58.23**±32.29	2.33±1.10
calc	4.65±0.75	**100**±0	**100**±0
xml	N/A	**86.20**±15.47	76.20±9.78
calcrs	N/A	**100**±0	**100**±0
jsonrs	N/A	**60.89**±12.74	56.21±11.31
calccpp	N/A	**100**±0	**100**±0
jsoncpp	N/A	**70.90**±10.45	66.07±8.23
xmlcpp	N/A	94.83±10.67	**95.92**±5.13

Takeaway 10

GDBMiner is more versatile than Mimid, yet achieves similar precision.

As a black-box approach, *Arvada* is easily applicable but delivers mixed precision results only. GDBMiner obtains the best precision on nine out of the eleven test programs. This is not surprising, as it is similar to the Mimid approach but much more versatile.

Takeaway 11

GDBMiner generates more precise grammars than Arvada.

Looking at the averaged recall results in Table 5.9, Arvada achieves best values on seven out of the eleven programs, while GDBMiner is mostly close behind and has best results on five. One outlier is GDBMiner on the *tinyc* program. Looking into the implementation, we can see that *tinyc* employs a *lexer* stage, which first translates input characters into predefined tokens. The actual parsing operates not the input buffer, but on the stream of tokens. Since GDBMiner can only track accesses to the input buffer, it looses track between input characters and their point of consumption in the program. Consequently, the derivation tree gets flawed, and the subsequent mining step can not generalize reasonably. Cmimid works around that limitation by explicitly following token compares on programs with preprocessing lexer stages, using

Table 5.10: F_1-scores in percentage averaged from 30 runs.

Program	Cmimid	Arvada	GDBMiner
cgidecode	**100**	96.70	**100**
json	68.49	73.68	**76.31**
mjs	72.85	70.78	**86.93**
tinyc	67.50	**66.71**	4.50
calc	8.88	99.99	**100**
xml	N/A	45.74	**85.96**
calcrs	N/A	**100**	**100**
jsonrs	N/A	67.82	**71.23**
calccpp	N/A	**100**	99.93
jsoncpp	N/A	74.50	**79.57**
xmlcpp	N/A	65.82	**96.17**

dynamic taint tracking. A generic taint tracking stage that works under our limited assumptions would be required to fix this inability of GDBMiner. However, only mining on a single of our case study programs would benefit from such a step, and therefore we keep GDBMiner's tracing stage simple and refrain from more complex analysis.

Compiling the averaged F_1-scores in Table 5.10, we can see that GDBMiner achieves the highest score on nine out of the eleven targets of our case study. Mined grammars with high F_1-scores signify that generated inputs are most likely valid but also cover a large portion and wide variety of available input features. Not surprisingly, we found that using more input seeds leads to higher recall values, because they have higher chances of covering more input features. Using 20 seeds, as done in [KLS21], seems to be a reasonable amount for all our case study programs, but an optimal value highly depends on the target program and diversity of the seeds.

Takeaway 12

GDBMiner yields most of the best accuracy scores.

The time needed to mine the grammars is rather unimportant as long as it remains within a usable frame. Nevertheless, we would like to concede that Arvada and Cmimid require only a few minutes for the runs, while GDBMiner can take several hours. This is taken to the extreme when mining on embedded hardware, which is why we will examine the temporal behavior closer in the next section.

Table 5.11: Performance of GDBMiner on embedded hardware.

Program	Precision	Recall	Tracing (hh:mm:ss)	Mining Time (hh:mm:ss)
cgidecode	93.9%	97.5%	01:09:50	00:00:48
json	99.3%	88.1%	03:29:14	00:02:24
xml	99.4%	93.5%	33:35:21	01:12:37

5.3.3 Evaluation on Embedded Programs

Next, we evaluate GDBMiner under the restricted conditions of *embedded systems*. We therefore choose the *B-L475E-IOT01A Discovery kit for IoT* from *STMicroelectronics* as a representative platform. It features an ARM Cortex-M4 core, several sensors and interfaces, six hardware breakpoints, four hardware data watchpoints, and an on-board debug probe. For our case study, we build programs for evaluating GDBMiner on embedded hardware using the cross-platform framework platformIO [Kra14]. We use libraries from its dependency management system for parsing *cgidecode* [Ota23], *json* [Ard22b], and *xml* [Kak21] data. The applications fetch input via the serial interface, run the corresponding parser functions, and return zero if the input was valid or a negative number else.

Tracing on embedded hardware with a limited amount of hardware watchpoints is significantly slower than on Linux applications mainly from three reasons:

1. The ARM Cortex-M4 core on our embedded hardware runs at a frequency of 120 MHz compared to 4.20 GHz of the Intel Xeon Gold 6144 we used in the experiments before.

2. The overhead of the debugging mechanism itself. GDBMiner needs to transfer each debug command first via TCP to a GDB server application, which transmits it via USB to the debug probe on the development board, and finally the debug probe forwards commands via the SWD (Single Wire Debug) protocol to debug unit on the processor.

3. The limited amount of hardware watchpoints forces us to trace each seed input multiple times. The exact number of traces we need to exercise per seed is determined by the length of the seed divided by the number of available watchpoints. Tracing a seed with 20 characters and the four available watchpoints requires five repetitions.

Again, we generate 20 seed inputs from the golden grammar for each test program, and calculate precision and recall values of the resulting grammar from 1,000 evaluation inputs. Additionally, we measure the time elapsed for tracing and mining. Table 5.11, shows that GDBMiner achieves high precision and recall values across all programs from our case study within a reasonable amount of time, even under the challenging conditions on our embedded hardware.

Takeaway 13

GDBMiner effectively mines input grammars even when the number of hardware watchpoints is limited.

5.3.4 Full-stack case study

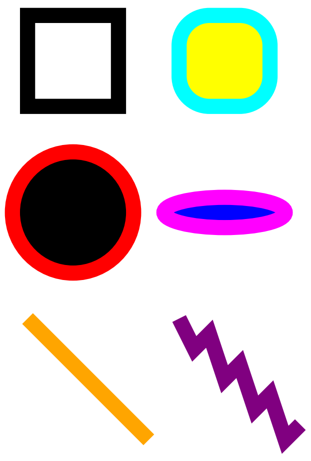
Finally, we go through the steps necessary to run GDBMiner on a real-world program and show what kind of analysis we can do with the resulting grammar. Industry extensively uses open source programs and to make all steps reproducible we will also demonstrate GDBMiner on an open source program. We want to emphasize again that GDB is available for most platforms and architectures, and hence these steps work on any program with a parser. The SVG++ library [Max14] processes Scalable Vector Graphics (SVG) images using different XML parser backends and includes an application that renders given vector images into a pixel-based image format. The followings steps are required to apply GDBMiner:

1. Configure to build SVG++ in debug mode.
2. Looking at the main function of the render application and ensuring that it reads input data from a char buffer.
3. Adopt a demo SVG image from Mozilla's SVG docs [MDN23] as seed, depicted in Figure 5.12a.
4. Define entry and exit points by source file line numbers and the char buffer name.

On our laptop with an Intel i7-10610U CPU, tracing and mining took about 14 hours. The resulting grammar is compatible with the fuzzingbook's grammar fuzzer [Zel+19]. A collection of generated images is shown in Figure 5.12b to visually express the diversity possible with grammar-generated inputs.

The grammar allows us to perform the following types of static and dynamic program analysis.

1. We can generate an unlimited number of random, but valid SVG images to test the SVG renderer looking for inputs causing timeouts or crashes. Indeed, many of our generated inputs cause hangs when defined shape sizes are too huge. A robust render application should cope with any input image, in our opinion, which is why we consider this behavior as buggy. For the following analysis, we alter the grammar to let numbers consist of one to three digits, only.
2. We analyze the grammar alone and reveal that any attribute within a node can be redefined arbitrarily times, such as multiple `fill=<color>` for the same shape node. The alternative XML parser backend of SVG++ does not allow attribute redefinitions, resulting in a different grammar. However, it is the programmer's decision whether this poses valid behavior.


(a) SVG seed image.

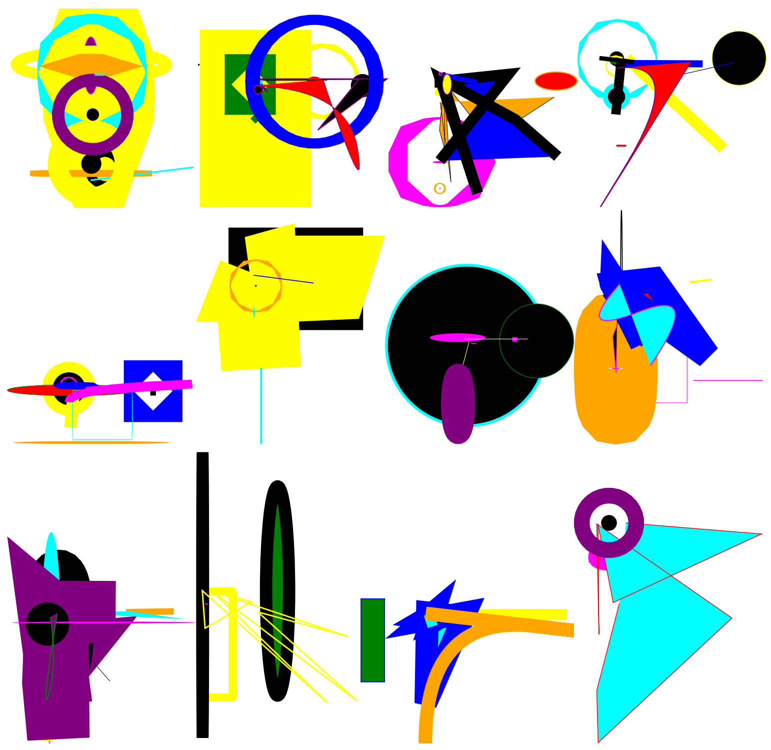

(b) Fuzzed images from mined grammar.

Figure 5.12: Mining and fuzzing a grammar of an SVG renderer.

3. We also compare the result of various SVG renderers to reveal differences. Again, we found that the default XML parser of the SVG++ library accepts redefinitions of node attributes, while the alternative XML backend, as well as Mozilla Firefox's SVG parser throw an error on redefinitions.

5.3.5 Threats to Validity

Like any empirical study, ours is subject to threats to validity.

External validity refers to the generalizability of research findings beyond the specific study context. While our subjects cover a number of features in input languages, they in no way can be representative for all input languages used—a data set that is not known. We think, however, that the restrictions placed by Cmimid on the C source code are unlikely to be met by most C programs; and we also think that tools like Arvada, which do not take the input-processing program into account, are by construction limited in their ability to infer a source language. While our results may not be representative, they at least confirm these assumptions.

Internal validity refers to the degree to which a study provides causal conclusions about the relationship between variables. To minimize the risk of systematic errors, we verified that our miners produce the required results on a small set of sample programs before applying it to the full set of targets.

Construct validity refers to the extent to which the variables and measurements accurately represent the underlying theoretical constructs. For accuracy, we use the standard measures of precision and recall that have been used in the literature on grammar mining before [HZ17; GMZ20; KLS21]. As it comes to *readability,* we are not aware of established metrics that would capture the readability of grammars; we leave it to the readers to decide whether they can comprehend the structure.

5.4 Conclusion and Future Work

In this chapter, we presented a practical debugger-driven grammar mining approach, called GDBMiner, which is able to derive context-free input grammars from any system that can be debugged with GDB. We explained how we leverage the debugger single-step functionality to trace programs and use limited amounts of hardware watchpoints to trace input buffer accesses. Additionally, we explained how we recover control flow graphs, reveal control flow structures, and recover derivation trees from just the obtained traces. Finally, we showed that we can transform the recovered derivation trees to receive human-readable and highly precise input grammars. GDBMiner generates near-perfect precise grammars and compared to state-of-the-art approaches reaches mostly higher accuracy (F_1) scores. We achieve similar results on embedded hardware using GDBMiner, as well. We find that GDBMiner is a versatile solution to mine precise input grammars from programs small and large, and recommend considering GDB as a robust and unified interface to a large variety of programs and architectures.

Potential future work could focus on the following topics:

Handle tokenizers. On one of our eleven case study programs, the point of input byte consuming mismatches the point of reading it due to a tokenizing stage. Static or dynamic mechanisms could detect such copying, and thus track bytes accurately in such cases regardless.

Binary formats. Knowing the proper syntax for input formats is helpful for fuzzing; yet, there are also *semantic constraints* to be fulfilled. Further research is required to infer higher order input specifications like ISLa [SZ22], which additionally specify semantic constraints across inputs, and in turn allow generating semantically valid inputs.

Speedup. GDBMiner is slow, requiring several hours to mine a grammar. Even when if the resulting grammar can generate myriads of test inputs rapidly efficiently, and therefore easily justifies the initial investment in CPU time, future work could investigate means to speed up GDBMiner without sacrificing versatility, notably by leveraging advanced tracing capabilities of modern microprocessors.

With the presentation of GDBMiner, we fulfil the industrial demand of versatile and accurate test data generators.

6 Closing

After discussing scientific contributions to the research community in the previous chapters, this final chapter concludes the dissertation. It encapsulates the main outcomes, highlights the implications of our research, and finally outlines potential future work lying ahead.

6.1 Conclusion

Practical application of fuzzing on embedded systems suffers from major challenges and impediments, as exhaustively explained in this thesis. We learned that:

- Cross-compiling embedded code for fuzzing is efficient – for *hardware-independent code.*
- Hardware-tracing approaches are powerful – if the hardware features *tracing capabilities.*
- Emulation-based fuzzing works great – given the availability of an appropriate *emulator.*
- Peripheral modeling automates emulation – presently for rather simple *peripheral devices.*

For all other cases, there were no easily applicable methods for fuzzing embedded systems prior to ours.

We analyzed the features and characteristics of embedded systems that can support us in establishing efficient automated software testing methods on the hardware itself. We identified single-stepping, along with setting hardware breakpoints and watchpoints, as the bare minimum of analysis methods available on most microcontrollers. Additionally, we find that these debug features can be controlled in an abstract way via the GDB remote serial protocol. Stripping down our requirements to just this limited set of features, allowed us to develop novel approaches that enable fuzzing embedded systems on a scale. We call it *debugger-driven embedded fuzzing.*

- GDBFuzz demonstrates that coverage-guided fuzzing on embedded systems is possible by extracting partial code coverage from hardware breakpoints. Already a single breakpoint can beneficially guide fuzzing and the use of dominator relations lowers interrupt overheads.
- GDBMiner enables learning of highly precise input specifications as context-free grammars on any system by tracing the execution with single-stepping and data watchpoints. Extracted context-free grammars enable grammar-based fuzzing but also serve for verifying input specifications.

Both approaches operate on the machine code level, which implicates that missing source code is not an obstacle, and they are programming language independent. Also, both approaches would be deployable in modern *continuous integration* and *continuous delivery* pipelines to allow *continuous fuzzing* of embedded systems during software development. We are therefore convinced that both methods can lead to more robust and secure embedded devices in the future. For transparency, and to foster further research, all code and raw experiment data is publicly available under:

`https://github.com/boschresearch/gdbfuzz`
`https://github.com/boschresearch/gdbminer`

6.2 Future Work

This thesis aims to establish the field of debugger-driven embedded fuzzing with lots of remaining potential for future research beyond GDBFuzz and GDBMiner. For now, we learned that:

1. GDBFuzz aims to maximize reached code coverage with the generated inputs using a feedback loop.
2. GDBMiner learns input grammars from a program under test, which can subsequently generate arbitrary amounts of valid test inputs without needing a feedback loop.

Both methods leverage only generic debug primitives, provided by GDB, allowing effective deployment on most embedded systems.

The question arises whether *coverage-guided fuzzing* and *learning input specifications* are orthogonal to each other. There are several publications that address both research areas: For instance, Godefroid *et al.* [GPS17] showed that code-coverage feedback can guide the learning of a recurrent neural network to generate diverse and valid PDF files. Moreover, Le *et al.* [Le+21] demonstrate how applying mutations to grammars can enhance their quality. Furthermore, Blazytko *et al.* [Bla+19] present an approach to infer generalized input specifications during coverage-guided fuzzing, and in turn leverage the input specifications to generate fuzzing inputs. The learning and the fuzzing task can therefore mutually benefit from each other. Consequently, a combination of GDBFuzz and GDBMiner might be an obvious way to further improve testing of systems.

There are essentially two possible straightforward combinations:

1. Use learned grammars to generate better test cases for coverage-guided fuzzing.
2. Use coverage-guided fuzzing to enhance the set of seed inputs for grammar learning or the learning step itself.

The use of context-free grammars to enhance coverage-guided fuzzing was shown by Aschermann *et al.* [Asc+19]. On top of the common mutation-based input generation, their approach generates inputs using the grammar and performs *grammar-aware* minimization and mutations on inputs. An example for a grammar-aware mutations is swapping matching subtrees from the parse trees of different seed inputs. A similar extension to GDBFuzz using learned grammars from GDBMiner would require little engineering efforts. A mutually supportive method like in [Bla+19] might also be feasible.

Let us careful think through the second option to combine GDBFuzz and GDBMiner:

1. GDBFuzz grows an input corpus by exploring the input space of the program under test over time.
2. GDBMiner could reuse this input corpus, as it depends heavily upon seeds that already cover large parts of the valid input space.

In practice, however, it is easier to derive an invalid input from a valid one using mutations than vice versa, which is why GDBFuzz mostly generates invalid inputs. Triggering the error handling code of a program under test increases the code coverage, which additionally guides GDBFuzz into generating invalid inputs. It is therefore unlikely, but not impossible, that GDBFuzz can improve a set of seed inputs, such that GDBMiner can learn a more generic grammar. Obviously, the chances of success depend heavily on the quality of the initial seeds in this case. Symbolic or concolic approaches may drastically improve chances of reaching new input language features, but this is beyond the scope of this work.

The thought experiment of synthesizing GDBFuzz and GDBMiner in different ways reveals the potential for cross-fertilization of ideas and methodologies. The synthesis of these approaches might open up new avenues for future research. The following – certainly incomplete – list is a collection of further proximate subtopics and coarse ideas of solutions.

Sanitization. Fuzzing relies on observable faults to find bugs. *Silent memory corruptions* are hard to detect, because the program does not run into an exceptional state that reflects the fault. Memory sanitization is an effective method for finding such memory faults. However, the required code instrumentation is not available or applicable for a variety of microcontrollers, in particular when source code is missing. A *debugger-driven memory sanitizer* might use hardware watchpoints to observe allocated memory regions on overflows. Similar to our presented approaches, such a sanitizer can not observe all memory buffers at once, but only a subset depending on the number of available watchpoints.

Differential Fuzzing. Invalid or unintended outputs of programs also do not manifest themselves via exceptions or interrupts. *Differential Fuzzing* leverages at least two different implementations of the same application and find differences between them while fuzzing. Different behavior or output might indicate implementation flaws. There are tons of protocols defined for embedded systems, most with multiple competing implementations, supplying lots of candidates for differential fuzzing. GDBFuzz can thereby generate test cases by coverage-guided fuzzing, or GDBMiner can learn an input model from one implementation to generate test inputs for all implementations.

Concolic Execution. A roadblock for coverage-guided fuzzing frequently occurs when parts of the input must exactly match predefined *magic values*. For instance, a USB CBW frame starts with `0x55 0x53 0x42 0x42`. Guessing such values correctly from scratch is unlikely. Symbolic execution can cope with such explicit values well, but quickly suffers from the path-explosion problem. Concolic execution selectively treats only parts of the inputs symbolically, to avoid too complex constraints. *Debugger-driven concolic execution* could augment debugger-driven fuzzing by applying symbolic execution on basic blocks that seem hard to reach i.e. the block was targeted often by GDBFuzz, but never reached.

Collecting the required constraints from the program in dependence of the respective input can be realized by a single-step pass, similar to how GDBMiner analyzes the system under test.

Stateful Fuzzing. Coverage-guided fuzzing expects the target program to behave deterministically, meaning that the same input always yields the same coverage-footprint. However, this might not be true if the system under test maintains a state. Trivially, we can reset or reboot the system after each test input or sequence of test inputs. However, this comes with an additional overhead and might prevent the fuzzer from reaching "deep" program states. *Stateful fuzzing* maintains awareness of the current state of the system and can therefore generate test cases more selectively. Learning state machines from a system under test could therefore lead to more effective fuzzing of stateful systems. The state of a system should be reflected somewhere in its memory, which is accessible via a debug probe. Again, we can leverage GDB to analyze the system's memory during its execution in order to find state reflecting variables. Observing these, while testing sequences of inputs would allow extracting a state machine.

Rich Input Specifications. Context-free grammars are great for describing repetitive and recursive patterns of a language. Nevertheless, these grammars are unable to represent context-sensitive properties such as check sums or length fields. Learning *richer input specifications* from the target program could enable generating massive amounts of valid inputs for all kinds of programs. Therefore, we require an even deeper analysis than that of GDBMiner. Instead of only extracting the control flow, we would need to grasp the constraints which lead to control flow decisions, similar to the earlier proposed debugger-driven concolic execution. These constraints could then augment context-free grammars mined with GDBMiner, for instance in the ISLa [SZ22] format.

Machine Learning. The number of proposals to use *machine learning* (ML) for software engineering and security automation skyrocketed in the past decade. Blindly following such hypes, however, may lead to premature conclusions regarding the effectiveness of such methods. In [NEZ23], we could show that a flavor of ML-guided fuzzing approaches, namely *neural program smoothing*, cannot be as effective as claimed from a theoretical point of view and also underperforms in practice. Nevertheless, there will certainly be areas where machine learning methods can enhance over traditional ones. For instance, choosing a promising subset of nodes to probe from the Control Flow Graph in GDBFuzz could be supported by ML.

As a final note, we hope that with our debugger-driven methodology we could open a door towards generic software analysis methods for embedded systems, and that future methods will continue to build on this foundation.

Happy Fuzzing!

Abbreviations

CFG	Control Flow Graph
DBI	Dynamic Binary Instrumentation
DMA	Direct Memory Access
DWT	Data Watchpoint and Trace
ETM	Embedded Trace Macrocell
FPB	Flash Patch and Breakpoint
FPGA	Field Programmable Gate Array
GDB	GNU Debugger
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
IoT	Internet of Things
JIT	Just-in-Time
JTAG	Joint Test Action Group
MMIO	Memory-Mapped IO
MMU	Memory Management Unit
MSC	Mass Storage Device Class
PLC	Programmable Logic Controller
SPI	Serial Peripheral Interface
SUT	System under Test
SWD	Serial Wire Debug
TCP	Transmission Control Protocol
VP	Virtual Prototype
SVG	Scalable Vector Graphics

Bibliography

[Aaf+21] Yousra Aafer, Wei You, Yi Sun, Yu Shi, Xiangyu Zhang, and Heng Yin. „Android SmartTVs Vulnerability Discovery via Log-Guided Fuzzing“. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021.

[ADG20] Chengwei Ai, Weiyu Dong, and Zicong Gao. „A Novel Concolic Execution Approach on Embedded Device“. In: *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy*. 2020, pp. 47–52.

[Agr94] Hiralal Agrawal. „Dominators, Super Blocks, and Program Coverage“. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1994, pp. 25–34.

[Agr99] Hira Agrawal. „Efficient Coverage Testing Using Global Dominator Graphs“. In: *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 1999, pp. 11–20.

[All70] Frances E Allen. „Control Flow Analysis“. In: *ACM Sigplan Notices* 5.7 (1970), pp. 1–19.

[Als19] Thomas Alsop. *Global Embedded Computing Market Revenue from 2018 to 2027 (in Billion U.S. Dollars)*. 2019. URL: `https://www.statista.com/statistics/1058799/worldwide-embedded-computing-market-revenue/` (visited on 03/09/2021).

[Ard22a] Arduino. *Arduino Default Fault Handler*. 2022. URL: `https://github.com/arduino/ArduinoCore-samd/blob/104f07f9053c49f60ceb0b09f9ae958fdee82cb8/cores/arduino/cortex_handlers.c#L28` (visited on 01/27/2022).

[Ard22b] Arduino Libraries. *Arduino_JSON*. 2022. URL: `https://registry.platformio.org/libraries/arduino-libraries/Arduino_JSON` (visited on 10/01/2023).

[ARM06] ARM. *ARM Debug Interface v5 Architecture Specification*. 2006. URL: `https://developer.arm.com/documentation/ihi0031/a/Overview-of-the-ARM-Debug-Interface-and-its-components` (visited on 05/05/2023).

[ARM21] ARM. *ARMv7-M Architecture Reference Manual*. 2021. URL: `https://developer.arm.com/documentation/ddi0403/ee/?lang=en` (visited on 05/05/2023).

[Asc+19] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. „NAUTILUS: Fishing for Deep Bugs with Grammars.“ In: *Proceedings of the Network and Distributed System Security Symposium*. 2019.

[ASU07] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Vol. 2. Addison-wesley Reading, 2007.

[Bas+17] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. „Synthesizing Program Input Grammars“. In: *ACM SIGPLAN Notices* 52.6 (2017), pp. 95–110.

[BCR21] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. „Fuzzing: Challenges and Reflections.“ In: *IEEE Softw.* 38.3 (2021), pp. 79–86.

[Bel05] Fabrice Bellard. „QEMU, a Fast and Portable Dynamic Translator“. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’05. Anaheim, CA: USENIX Association, 2005, p. 41.

[BF20] Marcel Böhme and Brandon Falk. „Fuzzing: On the Exponential Cost of Vulnerability Discovery“. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 713–724.

[BH19] Katharina Bogad and Manuel Huber. „Harzer Roller: Linker-Based Instrumentation for Enhanced Embedded Security Testing“. In: *Proceedings of the 3rd Reversing and Offensive-oriented Trends Symposium*. 2019, pp. 1–9.

[Bla+19] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. „GRIMOIRE: Synthesizing Structure while Fuzzing“. In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1985–2002.

[BLW21] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. „Estimating Residual Risk in Greybox Fuzzing“. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, pp. 230–241.

[Böh18] Marcel Böhme. „STADS: Software Testing as Species Discovery“. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27.2 (2018), pp. 1–52.

[Bör+20] Matthias Börsig, Sven Nitzsche, **Max Eisele**, Roland Gröll, Jürgen Becker, and Ingmar Baumgart. „Fuzzing Framework for ESP32 Microcontrollers“. In: *2020 IEEE International Workshop on Information Forensics and Security (WIFS)*. IEEE. 2020, pp. 1–6.

[BSM22] Marcel Böhme, László Szekeres, and Jonathan Metzman. „On the Reliability of Coverage-Based Fuzzer Benchmarking“. In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 1621–1633.

[Bui10] Franck Bui. *Implement basic parsers for parsing trivial arithmetic expressions*. 2010. URL: `https://github.com/fbuihuu/parser/blob/master/calc.c` (visited on 05/02/2023).

[C+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. „Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.“ In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[CCF18] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. „Inception: System-Wide Security Testing of Real-World Embedded Systems Software“. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 309–326.

[Che+16] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. „Towards Automated Dynamic Analysis for Linux-based Embedded Firmware.“ In: *Proceedings of the Network and Distributed System Security Symposium*. 2016.

[Che+18a] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. „IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.“ In: *Proceedings of the Network and Distributed System Security Symposium*. 2018.

[Che+18b] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. „Metamorphic Testing: A Review of Challenges and Opportunities“. In: *ACM Computing Surveys (CSUR)* 51 (2018).

[Che+19] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. „Enfuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers“. In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1967–1983.

[CHK01] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. „A Simple, Fast Dominance Algorithm“. In: *Software Practice & Experience* 4 (2001).

[Cle+20] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. „HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation“. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 1201–1218.

[Dav+13] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. „FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution“. In: *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 463–478. ISBN: 978-1-931971-03-4. URL: `https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson`.

[Def14] Defense Advanced Research Projects Agency (DARPA). *2014 Cyber Grand Challenge*. 2014. URL: `http://archive.darpa.mil/cybergrandchallenge/about.html` (visited on 11/13/2020).

[Del+20] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. „PHMon: A Programmable Hardware Monitor and Its Security Use Cases“. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 807–824.

[Din+20] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. „Retrowrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization“. In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1497–1511.

[Dol+15] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. „Repeatable Reverse Engineering with PANDA“. In: *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. PPREW-5. Los Angeles, CA, USA: Association for Computing Machinery, 2015. ISBN: 9781450336420.

[Dol+16] 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)*. 2016, pp. 110–121.

[DVD07] Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. „A Practical Interprocedural Dominance Algorithm“. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29.4 (2007), 19–es.

[Ear70] Jay Earley. „An Efficient Context-Free Parsing Algorithm“. In: *Communications of the ACM* 13.2 (1970), pp. 94–102.

[Edd31] AS Eddington. „Preliminary Note on the Masses of the Electron, the Proton, and the Universe“. In: *Mathematical Proceedings of the Cambridge Philosophical Society*. Vol. 27. 1. Cambridge University Press. 1931, pp. 15–19.

[Eis+22] **Max Eisele**, Marcello Maugeri, Rachna Shriwas, Christopher Huth, and Giampaolo Bella. „Embedded fuzzing: a review of challenges, tools, and solutions“. In: *Springer Cybersecurity* (2022).

[Eis+23] **Max Eisele**, Daniel Ebert, Christopher Huth, and Andreas Zeller. „Fuzzing Embedded Systems Using Debug Interfaces“. In: *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. 2023.

[Eis+24] **Max Eisele**, Johannes Hägele, Christopher Huth, and Andreas Zeller. „GDBMiner: Mining Precise Input Grammars on (almost) any System“. In: *Under Submission*. 2024.

[Eis22a] Max Eisele. *Buffer Overflow in cy_json_parser.c*. 2022. URL: `https://github.com/Infineon/connectivity-utilities/issues/2` (visited on 05/05/2023).

[Eis22b] **Max Eisele**. „Debugger-driven Embedded Fuzzing“. In: *2022 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. **(Best Submission Award)**. 2022.

[Eis22c] Max Eisele. *Infinite Loop in STM32 SCSI Driver*. 2022. URL: `https://github.com/STMicroelectronics/STM32CubeL4/issues/69` (visited on 05/05/2023).

[Eis22d] Max Eisele. *Null Pointer Dereference in cy_json_parser.c*. 2022. URL: `https://github.com/Infineon/connectivity-utilities/issues/1` (visited on 05/05/2023).

[Esp16] Espressif. *ESP32-DevKitC V4.* 2016. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html (visited on 02/15/2023).

[Fah17] Björn Fahller. *A variant of recursive descent parsing.* 2017. URL: https://github.com/rollbear/variant_parse (visited on 05/02/2023).

[Fas+21] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. „SoK: Enabling Security Analyses of Embedded Systems via Rehosting“. In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security.* 2021, pp. 687–701.

[Fen+21] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. „Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference“. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security.* 2021, pp. 337–350.

[Fio+20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. „AFL++ Combining Incremental Steps of Fuzzing Research“. In: *Proceedings of the 14th USENIX Conference on Offensive Technologies.* 2020, pp. 10–10.

[FML20] Bo Feng, Alejandro Mera, and Long Lu. „P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling“. In: *29th USENIX Security Symposium (USENIX Security 20).* 2020, pp. 1237–1254.

[FPH20] Rong Fan, Jianfeng Pan, and Shaomang Huang. „ARM-AFL: Coverage-Guided Fuzzing Framework for ARM-Based IoT Devices“. In: *International Conference on Applied Cryptography and Network Security.* Springer. 2020, pp. 239–254.

[Fre22] FreeRTOS. *Debugging Hard Fault & Other Exceptions.* 2022. URL: https://www.freertos.org/Debugging-Hard-Faults-On-Cortex-M-Microcontrollers.html (visited on 01/27/2022).

[Gar+20] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. „Certified Side Channels“. In: *29th USENIX Security Symposium (USENIX Security 20).* 2020, pp. 2021–2038.

[Gar23] Gartner. *Personal computer (PC) vendor shipments worldwide from 2006 to 2022 (in million units).* 2023. URL: https://www.statista.com/statistics/267023/global-pc-shipments-since-2006-by-vendor/ (visited on 05/02/2023).

[Gat99] Bill Gatliff. „Embedding with GNU: the GDB remote serial protocol“. In: *Embedded Systems Programming* 12 (1999), pp. 108–113.

[GLM12] Patrice Godefroid, Michael Y Levin, and David Molnar. „SAGE: Whitebox Fuzzing for Security Testing“. In: *Communications of the ACM* 55.3 (2012), pp. 40–44.

[Gmb19] SEGGER Microcontroller GmbH. *User guide of the J-Link application program interface (API).* Feb. 2019.

[GMZ20] Rahul Gopinath, Björn Mathis, and Andreas Zeller. „Mining Input Grammars from Dynamic Control Flow“. In: *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 2020, pp. 172–183.

[God20] Patrice Godefroid. „Fuzzing: Hack, Art, and Science“. In: *Communications of the ACM* 63.2 (2020), pp. 70–76.

[Goo16a] Google. *Fuzzer Test Suite*. 2016. URL: `https://github.com/google/fuzzer-test-suite` (visited on 11/22/2022).

[Goo16b] Google. *OSS-Fuzz*. 2016. URL: `https://google.github.io/oss-fuzz/` (visited on 05/20/2023).

[Goo22] Google Scholar. *Top 20 Computer Security & Cryptography Conferences*. 2022. URL: `https://scholar.google.com/citations?view_op=top_venues&vq=eng_computersecuritycryptography` (visited on 01/01/2022).

[GPS17] Patrice Godefroid, Hila Peleg, and Rishabh Singh. „Learn&fuzz: Machine Learning for Input Fuzzing“. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 50–59.

[Gre+12] Joseph L Greathouse, Hongyi Xin, Yixin Luo, and Todd Austin. „A Case for Unlimited Watchpoints“. In: *ACM SIGPLAN Notices* 47.4 (2012), pp. 159–172.

[Gro11] SystemC - System C Standardization Working Group. *IEEE 1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual*. 2011. URL: `https://standards.ieee.org/standard/1666-2011.html`.

[Gro20] Samuel Groß. *TrapFuzz*. Google Project Zero, 2020. URL: `https://github.com/googleprojectzero/p0tools/tree/master/TrapFuzz` (visited on 05/05/2023).

[Gue17] Guedou. *Using Miasm to Fuzz Binaries with AFL*. 2017. URL: `https://guedou.github.io/talks/2017%5C_BeeRump/slides.pdf` (visited on 01/01/2022).

[Gui+20] Zhijie Gui, Hui Shu, Fei Kang, and Xiaobing Xiong. „FIRMCORN: Vulnerability-Oriented Fuzzing of IoT Firmware via Optimized Virtual Execution“. In: *IEEE Access* 8 (2020), pp. 29826–29841.

[Gus+19] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. „Toward the Analysis of Embedded Firmware through Automated Re-hosting“. In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 2019, pp. 135–150.

[Har+20] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. „PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation“. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 789–806.

[Hea02] Steve Heath. *Embedded Systems Design.* Elsevier, 2002.

[HEC20] Shaobo He, Michael Emmi, and Gabriela Ciocarlie. „ct-fuzz: Fuzzing for Timing Leaks“. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST).* IEEE. 2020, pp. 466–471.

[Hel13] Yoran Heling. *Yxml - A small, fast and correct* XML parser.* 2013. URL: `https://dev.yorhel.nl/yxml` (visited on 05/02/2023).

[Her+19] V. Herdt, D. Große, H. M. Le, and R. Drechsler. „Early Concolic Testing of Embedded Binaries with Virtual Prototypes: A RISC-V Case Study*“. In: *2019 56th ACM/IEEE Design Automation Conference (DAC).* 2019, pp. 1–6.

[Her+20] Vladimir Herdt, Daniel Große, Jonas Wloka, Tim Güneysu, and Rolf Drechsler. „Verification of Embedded Binaries using Coverage-guided Fuzzing with SystemC-based Virtual Prototypes“. In: *Proceedings of the 2020 on Great Lakes Symposium on VLSI.* 2020, pp. 101–106.

[HMU01] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. „Introduction to Automata Theory, Languages, and Computation“. In: *Acm Sigact News* 32.1 (2001), pp. 60–65.

[HN16] Jesse Hertz and Tim Newsham. *TriforceAFL.* 2016. URL: `https://github.com/nccgroup/TriforceAFL` (visited on 02/09/2021).

[HZ17] Matthias Höschele and Andreas Zeller. „Mining Input Grammars with AUTOGRAM“. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C).* IEEE. 2017, pp. 31–34.

[HZ19] Nikolas Havrikov and Andreas Zeller. „Systematically Covering Input Structure“. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE. 2019, pp. 189–199.

[IC 22] IC Insights. *Microcontroller unit (MCU) shipments worldwide from 2015 to 2021 (in billions).* 2022. URL: `https://www.statista.com/statistics/935382/worldwide-microcontroller-unit-shipments/` (visited on 05/02/2023).

[IEE13] IEEE Computer Society. „IEEE Standard for Test Access Port and Boundary-Scan Architecture“. In: *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)* (2013), pp. 1–444. DOI: `10.1109/IEEESTD.2013.6515989`.

[Inf18] Infineon. *CY8CKIT-062-WiFi-BT PSoC 6 WiFi-BT Pioneer Kit.* 2018. URL: `https://www.infineon.com/cms/en/product/evaluation-boards/cy8ckit-062-wifi-bt/` (visited on 02/15/2023).

[Int17] International Organization for Standardization. „ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary“. In: *ISO/IEC/IEEE 24765:2017(E)* (2017), pp. 1–541. DOI: `10.1109/IEEESTD.2017.8016712`.

[Int18a] International Electrotechnical Commission. *Secure product development lifecycle requirements.* Standard. Geneva, CH: International Electrotechnical Commission, Jan. 2018.

[Int18b] International Organization for Standardization. *Road vehicles - Functional safety.* Standard. Geneva, CH: International Organization for Standardization, Dec. 2018.

[Int21] International Organization for Standardization. *Road vehicles - Cybersecurity engineering.* Standard. Geneva, CH: International Organization for Standardization, Aug. 2021.

[Jet22] Jetbrains. *Embedded - The State of Developer Ecosystem in 2022 Infographic.* 2022. URL: `https://www.jetbrains.com/lp/devecosystem-2022/embedded` (visited on 04/24/2023).

[Joh+21] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. „Jetset: Targeted Firmware Rehosting for Embedded Systems". In: *30th USENIX Security Symposium (USENIX Security 21).* 2021, pp. 321–338.

[Kak21] Ioulianos Kakoulidis. *PlatformIO Yxml.* 2021. URL: `https://registry.platformio.org/libraries/julstrat/LibYxml` (visited on 10/01/2023).

[Kal06] Marcin Kalicinski. *C++ XML Parser.* 2006. URL: `https://rapidxml.sourceforge.net/` (visited on 05/02/2023).

[Kha20] Sultan Qasim Khan. *Microcontroller Readback Protection: Bypasses and Defenses.* 2020.

[Kim+19] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. „RVFuzzer: Finding Input Validation Bugs in Robotic Vehicles through Control-Guided Testing". In: *28th USENIX Security Symposium (USENIX Security 19).* 2019, pp. 425–442.

[Kim+20] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. „FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis". In: *Annual Computer Security Applications Conference 2020.* ACM. 2020.

[Kin76] James C King. „Symbolic Execution and Program Testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[KKM15] Karl Koscher, Tadayoshi Kohno, and David Molnar. „SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems". In: *9th USENIX Workshop on Offensive Technologies (WOOT 15).* 2015.

[Kle+18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. „Evaluating Fuzz Testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* 2018, pp. 2123–2138.

[KLS21] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. „Learning Highly Recursive Input Grammars". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE. 2021, pp. 456–467.

[Knu71] Donald E Knuth. „Top-Down Syntax Analysis". In: *Acta Informatica* 1 (1971), pp. 79–110.

[KPK14] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. „PROSPECT: Peripheral Proxying Supported Embedded Code Testing“. In: *Proceedings of the 9th ACM symposium on Information, computer and communications security.* 2014, pp. 329–340.

[Kra14] Ivan Kravets. *PlatformIO.* 2014. URL: `https://platformio.org/` (visited on 05/02/2023).

[Lae+18] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. „RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs“. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* IEEE. 2018, pp. 1–8.

[Lau22] Lauterbach. *Lauterbach Development Tools.* 2022. URL: `https://www.lauterbach.com` (visited on 11/22/2022).

[Le+21] Xuan-Bach D Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. „Saffron: Adaptive Grammar-based Fuzzing for Worst-Case Analysis“. In: *ACM SIGSOFT Software Engineering Notes* 44.4 (2021), pp. 14–14.

[Li+22] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. „μAFL: Non-intrusive Feedback-driven Fuzzing for Microcontroller Firmware“. In: *Proceedings of the 44th International Conference on Software Engineering.* 2022, pp. 1–12.

[Lin16] Linda_pp. *Simple JSON parser/generator for Rust.* 2016. URL: `https://crates.io/crates/tinyjson` (visited on 05/02/2023).

[Liy+23] Danushka Liyanage, Marcel Böhme, Chakkrit Tantithamthavorn, and Stephan Lipp. „Reachable Coverage: Estimating Saturation in Fuzzing“. In: *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23), 17-19 May 2023, Australia.* 2023.

[LLV15] LLVM. *libFuzzer - a library for coverage-guided fuzz testing.* 2015. URL: `https://llvm.org/docs/LibFuzzer.html` (visited on 11/22/2022).

[LLV22] LLVM. *Clang DataFlowSanitizer.* 2022. URL: `https://clang.llvm.org/docs/DataFlowSanitizer.html` (visited on 05/14/2023).

[LO96] Mike Loukides and Andy Oram. „Getting to know GDB“. In: *Linux Journal* 29 (1996).

[LZZ18] Jun Li, Bodong Zhao, and Chao Zhang. „Fuzzing: A Survey“. In: *Cybersecurity* 1.1 (2018), pp. 1–13.

[Man+19] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. „The Art, Science, and Engineering of Fuzzing: A Survey“. In: *IEEE Transactions on Software Engineering* 47.11 (2019), pp. 2312–2331.

[Mar21] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things.* Springer Nature, 2021.

[Mat+19] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. „Parser-Directed Fuzzing“. In: *Proceedings of the 40th acm sigplan conference on programming language design and implementation.* 2019, pp. 548–560.

[Max14] Oleg Maximenko. *SVG++ documentation.* 2014. URL: `http://svgpp.org/` (visited on 01/23/2024).

[McK98] William M McKeeman. „Differential Testing for Software“. In: *Digital Technical Journal* 10.1 (1998), pp. 100–107.

[MDN23] MDN contributors. *SVG Tutorial - Basic Shapes.* 2023. URL: `https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Basic_Shapes` (visited on 01/23/2024).

[Mer+21] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. „DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis“. In: *2021 IEEE Symposium on Security and Privacy (SP).* IEEE. 2021, pp. 1938–1954.

[MFS90] Barton P Miller, Louis Fredriksen, and Bryan So. „An Empirical Study of the Reliability of UNIX Utilities“. In: *Communications of the ACM* 33.12 (1990), pp. 32–44.

[MI13] Alexander B Magoun and Paul Israel. „Did you Know? Edison Coined the Term “Bug”“. In: *IEEE Spectrum* (2013). URL: `https://spectrum.ieee.org/did-you-know-edison-coined-the-term-bug`.

[Mic17] Zalewski Michal. *American Fuzzy Lop (AFL).* 2017.

[MSP20] Dominik Maier, Lukas Seidel, and Shinjo Park. „BaseSAFE: Baseband SAnitized Fuzzing through Emulation“. In: *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks.* 2020, pp. 122–132.

[Mue+18a] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. „Avatar 2: A Multi-target Orchestration Platform“. In: *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)* Vol. 18. 2018, pp. 1–11.

[Mue+18b] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. „What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices.“ In: *Proceedings of the Network and Distributed System Security Symposium.* 2018.

[Nag+21] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. „Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing“. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security.* 2021, pp. 351–365.

[Nat19] National Security Agency. *Ghidra.* 2019. URL: `https://ghidra-sre.org/` (visited on 12/20/2021).

[Nat22a] Roberto Natella. „Stateafl: Greybox Fuzzing for Stateful Network Servers“. In: *Empirical Software Engineering* 27.7 (2022), p. 191.

[Nat22b] National Security Agency. *Ghidra function hasNoReturn.* 2022. URL: `https://ghidra.re/ghidra_docs/api/ghidra/program/model/listing/Function.html#hasNoReturn()` (visited on 01/27/2022).

[ND15] Anh-Quynh Nguyen and Hoang-Vu Dang. „Unicorn: Next Generation CPU Emulator Framework“. In: Jan. 2015.

[NEZ23] Maria Irina Nicolae, **Max Eisele**, and Andreas Zeller. „Revisiting Neural Program Smoothing for Fuzzing“. In: *Proceedings of the 31th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).* 2023.

[NH19] Stefan Nagy and Matthew Hicks. „Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing“. In: *2019 IEEE Symposium on Security and Privacy (SP).* IEEE. 2019, pp. 787–802.

[NNP19] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. „DifFuzz: Differential Fuzzing for Side-Channel Analysis“. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE. 2019, pp. 176–187.

[Noe12] Tammy Noergaard. *Embedded Systems Architecture 2nd Edition, A Comprehensive Guide for Engineers and Programmers.* Newnes, 2012.

[Nol+20] Yannic Noller, Corina S Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. „HyDiff: Hybrid Differential Software Analysis“. In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE).* IEEE. 2020, pp. 1273–1285.

[NS07] Nicholas Nethercote and Julian Seward. „Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation“. In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[Oh+15] JinSeok Oh, Sungyu Kim, Eunji Jeong, and Soo-Mook Moon. „OS-less Dynamic Binary Instrumentation for Embedded Firmware“. In: *2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII).* IEEE. 2015, pp. 1–3.

[Oku09] Kazuho Oku. *A header-file-only, JSON parser serializer in C++.* 2009. URL: `https://github.com/kazuho/picojson` (visited on 05/02/2023).

[Ota23] Kazuki Ota. *Arduino Percent.* 2023. URL: `https://registry.platformio.org/libraries/dojyorin/percent_encode/` (visited on 10/01/2023).

[Pan+21] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. „SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask“. In: *2021 IEEE Symposium on Security and Privacy (SP).* IEEE. 2021, pp. 833–851.

[PBR16] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. „Model-based Whitebox Fuzzing for Program Binaries“. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* 2016, pp. 543–553.

[PBR20] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. „AFLNet: A Greybox Fuzzer for Network Protocols“. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2020, pp. 460–465.

[Per17] Joshua Pereyda. *boofuzz: Network Protocol Fuzzing for Humans*. 2017. URL: `https://pypi.org/project/boofuzz/` (visited on 05/05/2023).

[PF20] Sebastian Poeplau and Aurélien Francillon. „Symbolic execution with SymCC: Don't interpret, compile!“ In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 181–198.

[PQ95] Terence J. Parr and Russell W. Quong. „ANTLR: A Predicated-LL (k) Parser Generator“. In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810.

[PY08] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons, 2008.

[Rav19] Ole André V Ravnås. *FRIDA Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers*. 2019. URL: `https://frida.re/` (visited on 11/04/2021).

[Red+21] Nilo Redini, Andrea Continella, Dipanjan Das, Giulio De Pasquale, Noah Spahn, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. „DIANE: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices“. In: *42nd IEEE Symposium on Security and Privacy 2021*. 2021.

[Rol21] Rollbar. *The State of Software Code Report (2021)*. 2021. URL: `https://content.rollbar.com/hubfs/State-of-Software-Code-Report.pdf` (visited on 10/01/2023).

[Rug+20] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. „Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets“. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 19–36.

[S+88] Richard Stallman, Roland Pesch, Stan Shebs, et al. „Debugging with GDB“. In: *Free Software Foundation* 675 (1988).

[Sac22] Goldman Sachs. *Average number of lines of codes per vehicle globally in 2015 and 2020, with a forecast for 2025*. 2022. URL: `https://www.statista.com/statistics/1370978/automotive-software-average-lines-of-codes-per-vehicle-globally/` (visited on 05/02/2023).

[SB19] Philip Sperl and Konstantin Böttinger. „Side-Channel Aware Fuzzing“. In: *European Symposium on Research in Computer Security*. Springer. 2019, pp. 259–278.

[Sch+22a] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. „Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing“. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 1239–1256.

[Sch+22b] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. „Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing“. In: *31st USENIX Security Symposium (USENIX Security 22)* (Aug. 2022). URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski`.

[Sch+22c] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. „Nyx-Net: Network Fuzzing with Incremental Snapshots“. In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 166–180.

[Sch17] Harald Scheirich. *JsonParser*. 2017. URL: `https://github.com/HarryDC/JsonParser` (visited on 10/01/2023).

[Seg22] Segger. *Segger Debug & Trace Probes*. 2022. URL: `https://www.segger.com/products/debug-trace-probes/` (visited on 11/22/2022).

[Ser+12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. „AddressSanitizer: A Fast Address Sanity Checker“. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, p. 28.

[SI19] Statista estimates and IHS Markit. *Server unit shipments by technology worldwide from 2016 to 2022 (in millions)*. 2019. URL: `https://www.statista.com/statistics/934508/server-unit-shipments-by-technology-worldwide/` (visited on 05/02/2023).

[Sko11] Sergei Skorobogatov. „Fault attacks on secure chips: from glitch to flash“. In: *Design and Security of Cryptographic Algorithms and Devices* (2011).

[Slo92] Thomas O'Conor Sloane. *The Standard Electrical Dictionary: A Popular Dictionary of Words and Terms Used in the Practice of Electric Engineering*. Henley, 1892.

[Son+19] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. „PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary“. In: *Proceedings of the Network and Distributed System Security Symposium*. 2019.

[Sor+20] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. „Inputs From Hell“. In: *IEEE Transactions on Software Engineering* 48.4 (2020), pp. 1138–1153.

[SP21] Prashast Srivastava and Mathias Payer. „Gramatron: Effective Grammar-Aware Fuzzing“. In: *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*. 2021, pp. 244–256.

[Spe+21] Chad Spensky, Aravind Machiry, Nilo Redini, Colin Unger, Graham Foster, Evan Blasband, Hamed Okhravi, Christopher Kruegel, and Giovanni Vigna. „Conware: Automated Modeling of Hardware Peripherals“. In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 2021, pp. 95–109.

[Sri+19] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. „FirmFuzz: Automated IoT Firmware Introspection and Analysis“. In: *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*. 2019, pp. 15–21.

[Sta23] StatCounter. *Global market share held by operating systems for desktop PCs, from January 2013 to January 2023*. 2023. URL: `https://www.statista.com/statistics/218089/global-market-share-of-windows-7/` (visited on 05/02/2023).

[STM19] STMicroelectronics. *STLINK-V3 modular in-circuit debugger and programmer for STM32/STM8*. 2019. URL: `https://www.st.com/en/development-tools/stlink-v3set.html` (visited on 02/16/2023).

[STM20a] STMicroelectronics. *B-L4S5I-IOT01A Discovery kit for IoT*. 2020. URL: `https://www.st.com/en/evaluation-tools/b-l4s5i-iot01a.html` (visited on 02/15/2023).

[STM20b] STMicroelectronics. *STM32 USB Host Library*. 2020. URL: `https://github.com/STMicroelectronics/STM32CubeL4/tree/v1.17.1/Middlewares/ST/STM32_USB_Host_Library` (visited on 01/27/2022).

[STM21] STMicroelectronics. *STM32 USB MSC Standalone*. 2021. URL: `https://github.com/STMicroelectronics/STM32CubeL4/tree/v1.17.1/Projects/B-L475E-IOT01A/Applications/USB_Host/MSC_Standalone` (visited on 01/27/2022).

[STM22] STMicroelectronics. *Cortex-M4 Fault Handler*. 2022. URL: `https://github.com/STMicroelectronics/STM32CubeF4/blob/4aba24d78fef03d797a82b258f37dbc84728bbb5/Projects/STM32F411RE-Nucleo/Examples_LL/SPI/SPI_OneBoard_HalfDuplex_DMA/Src/stm32f4xx_it.c#L59` (visited on 01/27/2022).

[Swi16] Robert Swiecki. *honggfuzz - Security oriented software fuzzer*. 2016. URL: `https://honggfuzz.dev/` (visited on 11/22/2021).

[SZ22] Dominic Steinhöfel and Andreas Zeller. „Input Invariants“. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 583–594.

[Tal+18] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. „Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems“. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 291–307.

[TBM21] Dimitrios Tychalas, Hadjer Benkraouda, and Michail Maniatakos. „ICSFuzz: Manipulating I/Os and Repurposing Binary Code to Enable Instrumented Fuzzing in ICS Control Applications“. In: *30th USENIX Security Symposium (USENIX Security 21).* 2021.

[Tex13] Texas Instruments. *MSP430F5529 USB LaunchPad development kit.* 2013. URL: `https://www.ti.com/tool/MSP-EXP430F5529LP` (visited on 02/15/2023).

[TH02] Mustafa M Tikir and Jeffrey K Hollingsworth. „Efficient Instrumentation for Code Coverage Testing“. In: *ACM SIGSOFT Software Engineering Notes* 27.4 (2002), pp. 86–96.

[The22] The MITRE Corporation. *2022 CWE Top 25 Most Dangerous Software Weaknesses.* 2022. URL: `https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html`.

[Tri+22] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. „Fuzzing Hardware Like Software“. In: *31st USENIX Security Symposium (USENIX Security 22).* 2022, pp. 3237–3254.

[Ver08] Bart Vermeulen. „Functional Debug Techniques for Embedded Systems“. In: *IEEE Design & Test of Computers* 25.3 (2008), pp. 208–215.

[Vos17] Nathan Voss. *Fuzzing the Unfuzzable.* 2017. URL: `https://hackernoon.com/afl-unicorn-part-2-fuzzing-the-unfuzzable-bea8de3540a5` (visited on 02/25/2021).

[Wri+21] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. „Challenges in Firmware Re-Hosting, Emulation, and Analysis“. In: *ACM Computing Surveys (CSUR)* 54.1 (2021), pp. 1–36.

[Yan+11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. „Finding and Understanding Bugs in C Compilers“. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation.* 2011, pp. 283–294.

[Yu+19] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. „Poster: Fuzzing IoT Firmware via Multi-stage Message Generation“. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* 2019, pp. 2525–2527.

[Yun+18] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. „QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing“. In: *27th USENIX Security Symposium (Security 2018).* Aug. 2018.

[Yun+22] Joobeom Yun, Fayozbek Rustamov, Juhwan Kim, and Youngjoo Shin. „Fuzzing of Embedded Systems: A Survey“. In: *ACM Comput. Surv.* (May 2022).

[Zad+14] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. „AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems’ Firmwares.“ In: *Proceedings of the Network and Distributed System Security Symposium.* 2014.

[Zel+19] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2019. URL: https://www.fuzzingbook.org/.

[Zhe+19] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. „FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation“. In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1099–1114.

[Zho+21] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. „Automatic Firmware Emulation through Invalidity-guided Knowledge Inference“. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021.