Projektarbeit
Studiengang Master Informatik

Fuzzing Microcontroller Software in an Emulator

von

Daniel Ebert
65926

Betreuender Professor: Prof. Roland Hellmann

Einreichungsdatum: 07.07.2021

Ehrenwörtliche Erklärung

I confirm that this work is my own work and I have documented all sources and materials used.

Aalen, den July 7, 2021

Daniel Ebert

Abstract

The security of software running on microcontrollers, such as IoT devices, is a major concern among users and producers [1]. Fuzzing is a practical and effective technique to automatically discover vulnerabilities at scale [2]. In fuzzing, inputs are repeatedly generated, processed by the software under test (SUT), and inputs that crash the SUT are reported to the user [3].

Different types of fuzzers exist. Graybox fuzzers typically use coverage feedback to generate inputs that reach higher code coverage in the SUT. Fuzzing microcontroller software with graybox fuzzers is often difficult because the special built-in microcontroller debugging facilities typically do not support advanced dynamic analysis techniques, such as feedback-driven graybox fuzzing [4].

To enable graybox fuzzing, the microcontroller software can be run and tested in an emulator. This work presents the design and implementation of an emulator fuzzer framework that integrates a mutation-based coverage-guided graybox fuzzer into the simavr emulator [5]. In addition, this work includes exercises in a virtual machine. These exercises introduce the participants to fuzz test microcontroller software that runs in an emulator via the implemented emulator fuzzer framework.

Contents

1 Introduction

Microcontrollers are often used for Internet of Things (IoT) devices [1]. The increasing number of IoT devices increases the amount of potentially vulnerable software [6] [1]. Since these devices are accessible through the Internet, any attacker can interact with them and attack them. This is one of the reasons why the security of software running on these microcontrollers is a major concern among users and producers [1].

This is not just a problem with non-critical applications such as low-cost IoT devices for home automation. Microcontroller software is also used in critical applications. For example, due to the trend of connecting vehicles to the cloud for services such as over-the-air software updates or remote diagnostics, security concerns have sharply risen in the automotive industry [7]. This is a critical security concern, because vehicle functions such as braking and steering are controlled by software [8].

Despite increasing security concerns, security awareness, and increasing efforts to improve the resilience of software against software vulnerabilities, software vulnerabilities are still common. In fact, the number of publicly disclosed security vulnerabilities in IT systems reached an all-time high of over 18 thousand vulnerabilities in 2020 [9]. Over 10 thousand of these were categorized with a severity of 'HIGH' or 'CRITICAL'.

Software vulnerabilities must be discovered and fixed before an attacker can exploit them. Otherwise, not only does the targeted device suffer physical and digital damage [1], but the image of the company or product also suffers and, depending on the industry, the company may have to deal with potential lawsuits.

There are manual and automated methods to discover software vulnerabilities. An example manual method is source code review, where vulnerabilities are discovered 'by hand'. One disadvantage of manual methods is that these methods do not scale [10]. Automated methods, such as fuzzing or static analysis, are required to keep up with the increasing number of software systems and the increasing complexity of these software systems [10]. Manual methods are still important, because not all vulnerabilities can be found via automated methods. However, automated and manual methods should be used in tandem to efficiently discover as many vulnerabilities as possible [11].

Fuzzing usually requires human involvement only at the beginning, when the software under test (SUT) is prepared for the fuzzer, and at the end, when the discovered vulnerabilities are analyzed and fixed. There are projects that also try to automate the initial and final steps. For example, there are projects that automatically prepare a fuzzer for testing a SUT [12] and projects that auto-generate human-readable explanations or patches for bugs [13]. However, currently these only work for specific use-cases and are in an early experimental stage [13].

Today, fuzzing is one of the most promising, practical, and effective techniques to automatically discover vulnerabilities at scale [2]. In fuzzing, inputs are repeatedly generated,

processed by the SUT, and inputs that crash the SUT are reported to the user [3]. Fuzzing automatically discovers vulnerabilities and bugs, and can be used at scale. For example, Microsoft uses fuzzing on thousands of machines to test all interfaces where inputs can come from untrusted sources [14].

There are 3 main categories of fuzzing tools: blackbox, whitebox, and graybox [3]. Fuzzing tools are called fuzzers. Blackbox fuzzers have no knowledge of the internal structure of the SUT. They can only observe the input and output behavior of the SUT [3]. There are two primary strategies to generate inputs in blackbox fuzzing: generational and mutational [2]. When the generational strategy is used, the fuzzer generates inputs based on a specification or a template [3], such as a regular expression. With this strategy, inputs are generated from scratch [2]. On the other hand, mutational strategies modify previous inputs to generate new inputs [3]. Modifications are called mutations. An example mutation is flipping a bit at a random location.

Graybox fuzzers have some knowledge of the SUT and primarily use mutation-based input generation [2]. Typically, the SUT is instrumented with lightweight instrumentation that detects what program locations are executed by an input [2]. If a generated input increases the code coverage, i.e. the input executes a program location for the first time since fuzzing started, the input is marked as 'interesting' [3]. Only inputs marked as 'interesting' are selected and mutated for the generation of new inputs. Thus, such graybox fuzzers gradually reach and thus test deeper parts of the SUT [2].

Whitebox fuzzers use symbolic execution or a variant of symbolic execution such as concolic execution [3]. These techniques analyze the internal structure of the SUT to calculate the conditions that an input must fulfill to execute a path in the SUT [2].

Despite its effectiveness in finding bugs and the small amount of required human involvement, fuzzing is not commonplace. In addition to the general challenges of fuzzing, fuzzing software that usually runs on a microcontroller results in additional challenges.

For example, gray- or whitebox techniques are difficult in this case. Many modern microcontrollers have special built-in debugging facilities [4]. However, these facilities are mostly designed for development and debugging, and do not support advanced dynamic analysis techniques, such as the feedback-driven graybox fuzzing [4].

The SUT can be emulated so that the fuzzer can access all runtime information, thus enabling feedback-driven graybox fuzzing. Due to the large variety of microcontroller boards and peripherals, there are not always emulators for all boards and the wide range of peripherals.

An additional challenge is that existing fuzzers often lack support for microcontrollers, for example due to incompatible fuzzing interfaces. Most fuzzers are designed for fuzzing SUTs that run on a Linux OS. For example, AFL passes generated inputs from the fuzzer process to the SUT process via inter process communication such as via a file or via the stdin stream [15]. Low-power boards, such as Arduinos, typically do not use a Linux OS. Additionally, a fuzzers may require that the SUT is compiled with a specific compiler that adds instrumentation. This compiler may not support an instruction set architecture for microcontrollers such as AVR.

1.1 Related Work

The mutation-based, graybox, and coverage-guided fuzzer that is implemented in this project builds on top of state-of-the art techniques. Fuzzing was first introduced in 1990 by Miller *et al.* to test the reliability of UNIX tools [16]. Miller *et al.* used blackbox fuzzing. Blackbox fuzzers lack guidance, because the fuzzer cannot learn from previously generated inputs. This makes it unlikely to reach certain paths in the SUT [17]. High code coverage is beneficial in fuzzing, because a fuzzer can only find bugs in code that is executed.

Cunningham *et al.* introduced the idea to guide inputs from fuzzers [18] [3]. They discuss how a fuzzer can use knowledge from past inputs to guide the input generation towards generating inputs that execute a specific destination location in the SUT [3]. The user must specify this destination [3].

Coverage-guided fuzzers are based on this idea of guiding the input generation. However, instead of guiding them towards one user-specified destination, coverage-guided fuzzers guide the input generation towards all destinations that increase the code coverage [3].

AFL [17] is one of the first and one of the most prominent representatives of coverage-guided fuzzers [3]. When the SUT source code is compiled, AFL instruments every basic block of the binary executable [17]. This instrumentation notifies AFL what basic blocks are executed by an input. Generated inputs that increase the code coverage are added to a queue [19]. Inputs in the queue are mutated to generate new inputs. AFL repeatedly forks the SUT process before the SUT's main function is executed [19]. Each forked SUT process is provided with one generated input. AFL can pass the input from the fuzzer to the forked process in several ways, such as via the stdin stream.

Many fuzzers are based on AFL. These extend or modify various stages of the fuzzing workflow with the primary goal of reaching higher code coverage or more bugs in less time. For example, AFLFast identifies inputs in AFl's queue that execute program parts that are rarely executed. These identified inputs are more often selected and mutated to generate new inputs [20]. AFL++ extends AFL with a Custom Mutator API to incorporate more mutations [19] [21]. For example, AFL++ can use the mutator from libFuzzer [21]. MOPT uses particle swarm optimization to select mutations and what parts of an input should be mutated [22].

There are fuzzers for microcontroller software. With the blackbox fuzzers IoTFuzzer [23], Snipuzz [24], and boofuzz [25], the microcontroller software runs on a real device and is not emulated. These fuzzers run on a separate system, for example a Ubuntu desktop PC, and they send inputs from the fuzzer to the device running the SUT via TCP/IP. Gray- or whitebox techniques are difficult in this case because the debugging facilities built into microcontrollers typically do not support advanced dynamic analysis techniques [4].

Running the fuzzer on the microcontroller together with the SUT is a possibility, but that comes with its own set of challenges. For example, there may not be enough storage on the microcontroller [4]. A mutation-based fuzzer must store a large set of previous interesting inputs that can be mutated. Some microcontrollers have low memory. For example, the Arduino Mega 2560 that is used in this project has 8 kilobytes of RAM [26]. In addition, firmware is often distributed only as a binary and without source code [27], which makes it difficult to instrument the SUT or add a fuzzer.

Fuzzers can use a hybrid approach, where a part of the SUT is emulated and the rest of the SUT, for example peripherals, runs on a real device. Fuzzers of this category include AFLtar [27]. AFLtar makes use of AFL and avatar^2 [27]. Avatar^2 is used to emulate a microcontroller processor and it forwards peripheral interactions to the actual physical device [27] [28]. AFLtar is a graybox fuzzer. The emulated processor informs AFL which basic blocks are executed/emulated by the SUT [27].

Emulators such as simavr [5] or QEMU [29] can emulate a microcontroller. This includes the emulation of a processor and emulation of on-board peripherals. Several fuzzers have integrated AFL into QEMU. This includes FirmAFL [30], P2IM [1], TriforceAFL [31], and AFL-Unicorn [32]. These are all coverage-guided graybox fuzzers and they mainly differ in how input is passed from the fuzzer to the SUT.

FirmAFL uses AFL and emulates the SUT with QEMU [30]. The fuzzer runs outside of the emulation. Inputs are passed to the emulated SUT via system calls, such as system calls for network operations [30]. For this reason, FirmAFL requires POSIX-compatible firmware [30].

P2IM also uses AFL and QEMU [1]. They forward peripheral interactions to the fuzzer [1]. This means that when the emulated SUT wants to read data from a peripheral, the requested data is generated and returned by the fuzzer instead [1].

AFL-Unicorn [32] uses AFL and the Unicorn emulator [33]. Unicorn is a fork of QEMU [33]. Users of afl-unicorn specify a virtual address. Inputs from the fuzzer are written to a buffer that starts at the specified virtual address in the SUT's process [34]. The fuzzer Unicorefuzz is based on AFL-Unicorn and can fuzz Linux kernel modules [35].

AFL++ can fuzz SUT running in QEMU and Unicorn [36]. Inputs are passed via file or via the stdin stream. AFL++'s QEMU mode only supports Linux SUTs [36]. AFL++'s Unicorn mode supports non-Linux SUTs [36], but the user must write code that injects the input from the fuzzer into the emulation [37]. For example, like AFL-Unicorn, the input can be written to a fixed address in the SUT's process [37].

AFL++'s QEMU mode has integrated support for QASan (QEMU-AddressSanitizer) [36]. QASan is a sanitizer that can detect heap related memory issues such as heap buffer overflows [38].

Brandon Falk has designed and implemented an emulator and integrated a coverage-guided fuzzer [39]. Like FirmAFL, the SUT must be POSIX-compatible. When the SUT uses a system call to read from a user-specified file, instead of returning the content of this file, the fuzzer generates and returns an input.

1.2 Goal of this project

This paper presents the design and implementation of a framework to fuzz test SUTs that are emulated via the simavr emulator. The key features of this framework are:

- Inject input generated by the fuzzer into the emulator.
- Detect when the emulator/SUT needs to be reset to start a new run with a new input from the fuzzer.

- Resetting the emulator/SUT.
- Collecting the code coverage of executed inputs.
- Collecting further information, such as values in global variables that store the state of the SUT at runtime.
- A coverage-guided fuzzer that uses the collected information to prioritize previous inputs for the generation of future inputs.
- Detect errors, such as buffer overflows.
- A user UI, where errors found during fuzzing are presented to the user.

In addition, chapter 5 provides exercises for working with this fuzzing framework. The exercises introduce participants to fuzz test software that runs in an emulator.

1.3 Outline

This section outlines the remaining paper. Chapter 2 provides relevant background information for fuzzing (section 2.1) and the simavr emulator (section 2.2). After that, chapter 3 describes the design of the framework, i.e. the fuzzer and sanitizers that were integrated into the simavr emulator, as well as the User API and User UI. Chapter 4 describes the implementation of these components. The student exercises in chapter 5 provide exercises and example solutions for students on how to fuzz SUTs with the framework that was implemented and is based on the simavr emulator. The paper ends with an evaluation and conclusion in chapter 6.

2 Background

This chapter provides relevant background information on the simavr emulator and fuzzers that are mutation-based and coverage-guided. The fuzzer integrated into simavr is of this type. To date, mutation-based graybox coverage-guided fuzzing is one of the most practical, efficient, and effective approaches to fuzzing [40].

2.1 Fuzzing

Fuzzing is an automated software testing technique. During fuzzing, the software under test (SUT) is repeatedly executed with inputs generated by a fuzzer [3]. A *fuzzer* is a program that implements a fuzzing technique. While the SUT processes inputs, the SUT process is monitored for interesting program states, such as a crash, timeout, excessive memory usage, or a failed assertion [3].

Fuzzing can be used to test any application that can read and process an input. A SUT can be a single function, an entire operating system, or a distributed application running on several machines. In practice, fuzzing is typically most effective when applied to interfaces of standalone application where inputs can come from untrusted sources [41]. Fuzzing for example distributed applications is possible, but usually requires a more complex fuzzer setup [41].

An input is an abstract concept [42]. In general, an input is a sample from the *input space* [42]. The input space consists of all possible data that the SUT can take from an external source [42]. For example, an input can be the content of a buffer or a file. When fuzzing GUI applications, an input can be a sequence of button clicks.

Inputs can be passed from fuzzer to SUT in different ways. If the fuzzer and the SUT are two separate processes, the input can be passed via inter process communication such as via files, the SUT's stdin stream, sockets, shared memory, or pipes [19] [36]. On the other hand, if the fuzzer and the SUT are one process, the fuzzer can for example call a SUT function and pass the input via function arguments [43]. Fuzzing a SUT in an emulator provides more ways to pass inputs, because the fuzzer has more control over the SUT process via the emulator. These are discussed in the User API design section 3.1.

Different fuzzers monitor different interesting program states in the SUT. For example, graybox coverage-guided fuzzers monitor not only invalid program states, such as a crash, but also whether an input increases the code coverage. Code coverage information is used to generate inputs more intelligently. This is later described in section 2.1.2.

Sanitizers are error checkers that can be used to detect additional program states, such as buffer overflows, use-after-frees, or use of uninitialized memory [44]. Without sanitizers,

these additional program states may not always be noticed. For example, use of uninitialized memory in a C program usually does not crash the process. However, if this can happen, the SUT may no longer function correctly anymore on all possible inputs. Unlike unit tests, fuzzing can usually not test the SUT for functional correctness, unless this is achieved indirectly via e.g. assertions.

Fuzzers store inputs that result in interesting program states (e.g. a crash) in the SUT to disk. Often fuzzers also output additional information about the interesting program state, for example the program counter when the interesting program state was detected. This information is often not sufficient to fix the bug. The user can use the input to reproduce the bug and analyze the situation with a debugger [41]. To enable this reproducibility, any relevant application state in the SUT must be reset after the SUT has finished processing an input. This is usually achieved by either restarting the SUT process, forking the SUT process prior to calling the SUT's main function, or the SUT may implement logic that resets the relevant application state. Also note that not all SUTs have relevant application state.

2.1.1 Mutation-based Input Generation

There are several strategies for generating inputs. Most early fuzzers generated inputs randomly [3]. A random input is unlikely to satisfy specific path conditions [3]. Consider the following SUT:

```
1 void entry(uint8_t *input, size_t input_size) {
2   if (input_size > 0 && input[0] == 'H')
3     if (input_size > 1 && input[1] == 'I')
4       if (input_size > 2 && input[2] == '!')
5         crash();
6 }
```

Listing 2.1: SUT with string compare

If the *input* string is randomly generated, the probability of calling *crash()* is 1 in 256^3 (circa 16 million). The generated inputs are likely rejected in the input checks stage and will likely not reach deeper parts of the program. Fuzzing can find bugs only in code that is executed and therefore deeper parts of the program are not tested. The situation gets worse when the SUT expects well-structured input such as JSON [3].

Mutation-based input generation is a strategy that increases the chance to generate inputs that reach deeper parts of the SUT [45]. A *mutator* applies small modifications to existing inputs [45]. Modified inputs are generated inputs that are passed to the SUT.

This modification is referred to as a mutation [3]. The mutator must decide where to mutate, i.e. at what position in the input, and what mutation should be applied. Example mutations include flipping, adding, or removing a bit or byte. In addition to these mutations, AFL can also apply more intelligent mutations. For example, AFL can select 4 bytes from an input, interpret the selected bytes as an integer, and increment or decrement this integer [3].

An example mutation is shown in figure 2.1. In this example, the SUT expects JSON string inputs.

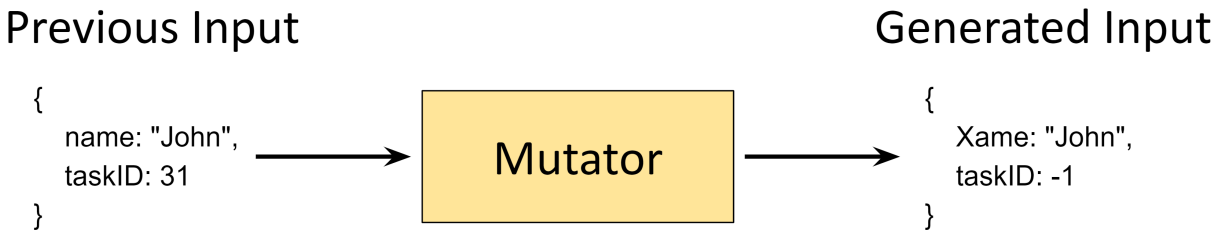

Figure 2.1: Mutator

The mutator typically applies multiple mutations to each generated input [42]. Mutations and input positions to mutate are often chosen randomly. Mutators are usually not aware of the expected input format or specifications [46].

The input to the mutator is usually an input that reaches deeper parts of the SUT. Thus, modified inputs are often valid enough to pass parsing checks and reach deeper parts of the program, yet exercise new behavior [45], because these deeper parts are tested with different inputs.

The user can provide a set of inputs. These user-provided inputs are called *seeds* [3]. Seeds are the inputs for the mutator. The mutator applies mutations to the seeds to generate inputs for the SUT.

2.1.2 Coverage-guided Fuzzing

Coverage-guided fuzzers mark generated inputs as 'interesting' if the input increases the code coverage [3]. Previous interesting inputs can, like seeds, be passed to the mutator to mutate the previous interesting input in order to generate inputs. Coverage-guided fuzzers gradually increase the code coverage and gradually reach deeper parts of the SUT [2].

The primary goal of fuzzers is to discover as many bugs or vulnerabilities in the SUT as possible within a limited time budget. Coverage-guided fuzzers aim to maximize code coverage to test as much functionality of the SUT as possible. A high code coverage is beneficial, because there is strong empirical evidence that a fuzzer with higher code coverage discovers more bugs in the SUT in a given time window [42] [40].

Coverage-guided fuzzers can start without seeds. However, for most SUTs this is not recommended, because it often takes time to build up a set of previous interesting inputs that reach a high code coverage or deeper parts of the SUT.

To collect code coverage, a piece of code is executed every time the SUT encounters an edge in the control flow graph [47]. In source code these are for example 'if' and switch statements. The injected code is called instrumentation. Instrumentation notifies the fuzzer what code is executed by an input. Collecting coverage for inputs comes with a performance overhead of typically 20% to 50%.

2.2 simavr

Simavr is an *emulator*. An emulator allows a process to behave like another system. With simavr, a process can behave like a microcontroller system [5]. In other words, simavr can imitate the operations of a microcontroller in software without needing to have a real microcontroller [48] [49]. A microcontroller contains one or multiple processor cores and on-board peripherals. There exist different cores and different on-board peripherals. Simavr can emulate various 8-bit AVR instruction set architecture cores, such as the ATMega2560, and various peripherals, such as IO ports and timers [5]. For example, simavr can emulate the Arduino ATMega2560 microcontroller [5].

The instructions of the executable machine code describe the behavior of the processor. Therefore, emulating the processor of a microcontroller consists of emulating the effects of the instructions of the system being emulated. These effects include, for example, interacting with the memory management unit (MMU) to read or write to registers and memory, and interacting with on-board peripherals [49].

The systems that the processor interacts with, such as the MMU and on-board peripherals, can be seen as the emulation environment. For this reason, an emulator for microcontrollers must emulate not only the processor core that processes instructions, but also the environment such as on-board timers that run independently of the processor core [49].

For this purpose, simavr sets up and makes use of a virtual MMU, virtual registers, and virtual on-board peripherals. The emulated machine code instructions change the emulated environment as the instructions of the system being emulated would do with the original system environment. Similarly, when the emulated environment interacts with the emulated processor, for example via an interrupt that is triggered when an on-board timer runs out, the emulated processor behaves as the system being emulated would.

AVR is a RISC architecture. Simavr's emulation of the processor consists of the typical stages in a RISC instruction pipeline [50], namely:

1. **Instruction Fetch:** The address pointed to by the value in the virtual program counter register is read from virtual memory [50].
2. **Instruction Decode and Execute:** In the simavr implementation, this is a large switch statement [51]. The opcode of the instruction specifies what branch is executed [51]. This branch emulates the effects of the instruction. For example, an 'add' instruction adds the values in two virtual registers [52].
3. **Write Back:** Lastly, the result is written to virtual registers [50].

After the initial setup of the emulator, simavr runs a so-called main loop. In each loop iteration the instruction steps are performed [51]. In addition, simavr also takes care of the emulated environment in each loop iteration [51]. This includes, for example, checking whether a timer has expired or whether an interrupt is pending and, if this is the case, servicing the interrupt [53]. These systems would typically run in parallel [49]. However, simavr, like most emulators, is single threaded. This is the case, because if every system would run in a separate thread, the synchronization between the different emulated systems would be difficult to implement [49].

The initial setup of the emulator mentioned in the previous paragraph consists of for example allocating memory for the virtual MMU, i.e. virtual registers and virtual RAM,

loading the emulated program data and code into this allocated memory, and setting up systems like the timer or interrupt systems [54]. The simavr implementation uses an 'avr' struct to store references (i.e. pointers) to e.g. the allocated memory for the MMU and structs that manage systems like the timer or interrupt systems.

The main loop runs until the emulated program terminates. Since microcontroller software typically does not terminate, the main loop is often an infinite loop.

3 Design

This chapter describes the design of the emulator fuzzer framework. This includes identifying and describing problems that need to be solved and discussing solutions for these problems.

3.1 User API

Microcontroller software can read input from different sources. For example, input can be read via the serial interface. In this case, the SUT may expect that the input is returned from the *Serial.read()* Arduino standard library function. Another SUT may require that the input is written to RAM via direct memory access (DMA) and that the DMA is followed by an external interrupt. A unit test may expect that the input is passed via one or multiple function arguments.

A user API is needed because the user must specify when, where, and how the input is passed from the fuzzer to the SUT. For example, the user can configure that *Serial.read()* returns input from the fuzzer or that the input from the fuzzer is written to a buffer in RAM via direct memory access before the SUT reads from this buffer.

There are several alternatives for the design of the user API. For example, libFuzzer [43] provides one way to pass input to the SUT, namely that a specific function is called by the fuzzer and the input is passed to that function via function arguments. In this case, no complex user API is required for input passing because there is only one option. Due to the large variety in input sources in microcontroller software, a solution is required that offers multiple options from which a user can choose.

In addition to libFuzzer's function argument option, AFL [17] can also pass input via a file or via the stdin stream. The configuration is specified with a predefined flag. This flag is passed to AFL via command-line arguments [55]. Configuration via flags in command-line arguments or via environment variables is beginner-friendly. This advantage is particularly beneficial with regard to providing a less complicated introduction to fuzzing for students in the student exercises. One disadvantage of this approach is that configuration via command-line options or environment variables does not scale well as configurations become increasingly complex. In other words, adding more configuration options and special cases for these options results in less clarity.

In the chosen design, the API is accessed via the C language. A similar design is also used in other fuzzers such as boofuzz [56]. The difference is that boofuzz's API is accessed via Python instead of C [56].

The main reason why the C API approach was chosen is because this approach allows complex configurations. Furthermore, the emulator is written in C. The user can access

the functions and data structures used by the emulator itself via the C API. Thus, there is less of a need to implement abstractions for each emulated system that the user might want to work with. These systems include for example the memory management unit, the interrupt controller, various emulated peripherals and I/O, and others.

Compared to the command-line arguments approach, the C API is less beginner-friendly. To address this issue, helper functions (i.e. abstractions) for emulator data structures and functions are provided for the user. These abstractions are C functions. They simplify the setup that the user would otherwise have to program/configure. Helper functions exist for common use cases, including those outlined in the first paragraph of this section.

3.1.1 Patch Instruction and Direct Memory Access

The user must specify when, where, and how the input is passed from the fuzzer to the SUT. For this purpose, the C API provides the *patch_instruction* function:

```
1 int patch_instruction(avr_flashaddr_t vaddr,
2                        void *function_pointer,
3                        void *arg);
```

Listing 3.1: patch_instruction function prototype

Users make use of *patch_instruction* to specify that a function should be executed when the SUT is about to execute a particular instruction. The *vaddr* argument is the address of an instruction in the SUT's program space. Prior to emulating this instruction, the emulator calls the function pointed to by the *function_pointer* argument. Thereby the *arg* argument is passed as a parameter to the *function_pointer* function. The *function_pointer* function is run 'outside' of the emulation and has access to all emulator data structures, emulator functions, and API helper functions.

The following example 3.2 shows an exemplary use of the *patch_instruction* function:

```
1 void setup_patches(avr_t *avr) {
2   patch_instruction(0x123, callback_function, avr);
3 }
4
5 void callback_function(void *arg) {
6   avr_t *avr = (avr_t *)arg;
7   write_to_ram(0x456, "A", avr);
8 }
```

Listing 3.2: patch_instruction example

The emulator calls the *setup_patches* function during emulator initialization. Emulator initialization takes place before the SUT is emulated and thus before fuzzing starts. Users implement the *setup_patches* function. In example 3.2, the implementation consists of a call to *patch_instruction*. This specifies that the emulator process calls *callback_function* with *avr* as function argument when the SUT is about to execute the instruction at address *0x123*.

A pointer to an *avr_t* instance is passed to the *setup_patches* function. An *avr_t* instance contains the data that an emulator needs to emulate an AVR program. The *avr_t* instance is the 'avr' struct that was described in the simavr background section 2.2.

callback_function calls the *write_to_ram* function. *write_to_ram* is already implemented. It is one of the API helper functions for working with emulator data structures in *avr* that are provided for the user. *write_to_ram* provides an abstraction around the emulated MMU. It can be used to override data in the SUT's RAM, such as global variables. In this case, *A* is written to the SUT's RAM at address *0x456*.

The API helper function *write_to_ram* can also be used to overwrite function arguments and function return values. These are passed from caller to callee via registers and, if there are too many function arguments, via the stack [57] and simavr stores the register values in RAM.

3.1.2 Symbols and Writing to Global Variables

A common use case in the student exercises is to write the input and the input length to global variables in the SUT when the SUT calls a specified function. The following listing 3.3 shows a part of the source code of an example SUT used in the exercises:

```
1 char fuzz_input[256];
2 uint16_t fuzz_input_length;
3
4 void setup() {
5   parse(fuzz_input, fuzz_input_length);
6 }
7
8 void parse(...
```

Listing 3.3: SUT example with input via global variables

Both *fuzz_input* and *fuzz_input_length* are uninitialized global variables. The following *setup_patches* write the input and the input length to these global variables when the SUT calls the *setup* function:

```
1 void setup_patches(avr_t *avr) {
2   patch_instruction(get_symbol_address("setup", avr),
3                     write_fuzz_input_global,
4                     avr);
5 }
```

Listing 3.4: patch_instruction example for input via global variables

Listing 3.4 uses two API helper functions that are already implemented and provided to users. *get_symbol_address* returns the address of an ELF symbol with a given name. In this example, *get_symbol_address* returns the address of the first instruction of the SUT's *setup* function. The *write_fuzz_input_global* function fetches an input for the SUT from the fuzzer/mutator. This input is then written to the SUT's *fuzz_input* vari-

able. The length of this input is written to *fuzz_input_length*. The implementation of *write_fuzz_input_global* uses *get_symbol_address* and *write_to_ram*.

Participants of the exercises will typically use the *patch_function* function shown in listing 3.5. The *patch_function* function is a wrapper for the *patch_instruction* and *get_symbol_address* functions.

```
int patch_function(char *function_name, void *patch_pointer, void *arg,
                   avr_t *avr) {
  return patch_instruction(get_symbol_address(function_name, avr),
                           patch_pointer,
                           arg);
}
```

Listing 3.5: patch_function Function

```
void setup_patches(avr_t *avr) {
  patch_function("setup", write_fuzz_input_global, avr, avr);
}
```

Listing 3.6: patch_function example for input via global variables

3.1.3 External Interrupt

Users are provided with a C API helper function called *raise_external_interrupt* to trigger an external interrupt:

```
void raise_external_interrupt(uint8_t pin, avr_t *avr);
```

Listing 3.7: raise_external_interrupt Function

The argument *pin* specifies which via which GPIO pin the interrupt is received. For example, with the following SUT source code in listing 3.8 and the user API in listing 3.9, the interrupt at pin 19 is triggered when the SUT calls the loop() function. Due to the interrupt at this pin, the *my_interrupt_service_routine* function is called. The *pin* value is the same that is passed to the *digitalPinToInterrupt* function for *attachInterrupt* in the SUT source code.

```
void setup() {
  Serial.begin(9600);
  pinMode(19, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(19), my_interrupt_service_routine,
    CHANGE);
}

void my_interrupt_service_routine() {
  Serial.println("Handling IRQ.");
}

```

```
11 void loop();
```

Listing 3.8: SUT with Interrupt Example

```
1 void raise_interrupt(void *arg) {
2   avr_t *avr = (avr_t *)arg;
3   raise_external_interrupt(19, avr);
4 }
5
6 void setup_patches(avr_t *avr) {
7   patch_instruction(get_symbol_address("loop", avr), raise_interrupt, avr);
8 }
```

Listing 3.9: C API for the Interrupt Example

3.1.4 Integration

The source code with the user implemented setup_patches function is compiled to a shared library. This library is made available to the emulator at runtime via the environment variable LD_LIBRARY_PATH. The shared library approach was chosen in favor of compiling the setup_patches along with the rest of the emulator due to flexibility and speed. With LD_LIBRARY_PATH, users can quickly switch between different setup_patches implementations. Compiling a shared library is also faster compared to recompiling and linking the whole emulator code.

3.2 Sanitizer

A *sanitizer* is an error detector [58]. They track the execution of a process and report runtime errors when one is detected [58]. Sanitizers are important in software testing, because they can detect bugs that could not have been found otherwise.

Existing sanitizers were analyzed. However, none were compatible for an AVR use case. For this reason, sanitizers were implemented and integrated into simavr.

Several sanitizers already exist for different languages including C/C++. They can detect various types of errors. Google developed several sanitizers and they include, among others, the AddressSanitizer (ASAN) and MemorySanitizer [59]. ASAN detects memory corruption issues such as buffer overflows[59]. MemorySanitizer detects use of uninitialized memory [59].

Google's sanitizers are *compiler sanitizers* [58]. Compiler sanitizers instrument the SUT executable during compilation. This instrumentation implements the logic for detecting errors.

Google's sanitizers are built into GCC since version 4.8 [44]. However, these sanitizers require that the SUT is linked with an implementation of the C++ Standard Library. The avr-gcc (and avr-g++) compiler has no full C++ Standard Library support. For this

reason, Google's sanitizers do not work with avr-gcc. avr-gcc is used by the Arduino IDE and PlatformIO for AVR boards.

Google's sanitizers are also built into LLVM [60]. The clang compiler is the frontend for LLVM. Similarly to avr-gcc, LLVM does not support the C++ Standard Library for AVR targets [61]. In addition, the LLVM AVR implementation is marked experimental [62].

There are open-source projects that try to implement partial C++ Standard Library support for AVR [63] [64]. Partial C++ Standard Library support means that they implement a library that implements a part of the functions of the C++ Standard Library. These projects are not maintained anymore [63]. It is also unclear whether a partial C++ Standard Library is sufficient for the sanitizers.

Other sanitizers like Valgrind also do not support AVR [65]. As an alternative to sanitizers, there are also static analysis tools that try to find similar types of bugs. For example, GCC can be configured to detect the use of uninitialized memory. This option is enabled via the '-Wuninitialized' command line flag. GCC's goal is like MemorySanitizer's goal. However, GCC's static analysis can result in false positives and there are more false negatives compared to MemorySanitizer's dynamic code analysis [66].

Simavr had already implemented sanitizers for the following types of errors [51]:

- Invalid Opcodes
- Invalid read and write accesses
- Execution of non-existent instructions (e.g. resulting from invalid jumps)

Additional sanitizers have been implemented and integrated into simavr to detect more and different types of errors. Specifically, these additional sanitizers can detect the following types of errors:

- Stack Buffer Overflows
- Uninitialized Memory
- Timeouts

The design of these sanitizers is described in more detail in section 3.2.1, 3.2.2, and 3.2.3.

Sanitizers can detect more types of errors. For example, Google's sanitizers can also detect heap related issues [44]. This includes, among others, heap buffer overflows, double-frees, and use-after-frees [44]. Sanitizers for heap related issues were not implemented because the heap is not commonly used in Arduino/microcontroller software due to low available memory and therefore memory fragmentation can be a problem.

Google's Undefined Behavior Sanitizer (UBSAN) detects signed integer overflows [60]. Both the C and C++ Standards [67] specify that signed integer overflows are undefined. Unlike signed integer overflows, unsigned integer overflows are not undefined behavior [67]. For this reason, a sanitizer for integer overflows must be able to determine whether a computation is done with signed or unsigned integers.

Google sanitizer instrumentation, including UBSAN instrumentation, is implemented as a compiler optimization pass [68]. Optimization passes optimize/process a compiler intermediate representation (IR). Clang for example uses the LLVM IR [69]. The variables in LLVM IR have type information [69]. For this reason, UBSAN can differentiate between signed and unsigned integers.

The simavr emulator, on the other hand, works with AVR machine code. AVR machine code has no information about the variable types from the original source code. Based on AVR machine code alone, it is difficult to infer variable types. Thus, it is difficult to differentiate between signed and unsigned integers. Since this is a requirement for integer overflow sanitizers, no integer overflow sanitizer is integrated into simavr. Implementing a compiler optimization pass would go beyond the scope of this project.

Integer overflows are invalid in cases such as during a Linux system call that resulted due to invalid arguments from the SUT [70]. Sanitization for integer overflows can also be done in these cases. Sanitizers for this special case were not implemented, because Arduino software uses the Arduino Framework/Library and not an OS like Linux [71].

Google's Thread Sanitizer (TSAN) detects data races in multi-threaded processes [72]. Arduino does not support parallel tasks, i.e. threads. For this reason, no sanitization is implemented for data races.

3.2.1 Stack Buffer Overflow

Simavr has implemented a stack buffer overflow sanitizer. This sanitizer is disabled by default. Defining the AVR_STACK_WATCH preprocessor macro enables this sanitizer.

Simavr's stack buffer overflow sanitizer implementation detects and prints errors when the emulated program writes to a stack frame that is not the stack frame of the current function [73]. This design results in false positives because there are valid cases when the SUT must write to other stack frames.

For example, simavr with the AVR_STACK_WATCH option enabled prints an error when the following program 3.10 is compiled and emulated:

```
1 void setup() {
2   int i = 0;
3   plus_one(&i);
4 }
5
6 int plus_one(int *arg) {
7   (*arg) += 1;
8 }
```

Listing 3.10: simavr AVR_STACK_WATCH False Positive

Line 7 in listing 3.10 writes to memory of the previous stack frame. Simavr with the AVR_STACK_WATCH option enabled classifies this as a bug. This is a false positive.

False positives should be avoided, especially if the tool is to be used for learning, as in the student exercises. Students who are learning a new tool do not know if they or the tool made a mistake.

ASAN's stack overflow sanitizer implementation creates memory areas in between stack variables [69]. These memory areas are referred to as poisoned redzones [69]. Writing to poisoned redzones is considered invalid behaviour. ASAN adds instrumentation that check if the process writes to poisoned redzones [69]. This design detects both buffer

overflows and underflows [69]. ASAN uses a similar design to sanitize the heap and global variables for buffer over- and underflows [69].

ASAN's design requires instrumentation that creates the poisoned redzones. ASAN adds this instrumentation to IR during compile time. The ASAN paper states that it is unclear how this instrumentation can be implemented using run-time instrumentation [74]. There are several reasons for this. For example, it is difficult to identify stack variables and the size of these variables based on the machine code. This information is required to add poisoned redzones in between stack variables. Another reason is that AVR machine code uses constant offsets to address memory on the stack. These offsets must be adjusted when poised redzones are added in between stack variables.

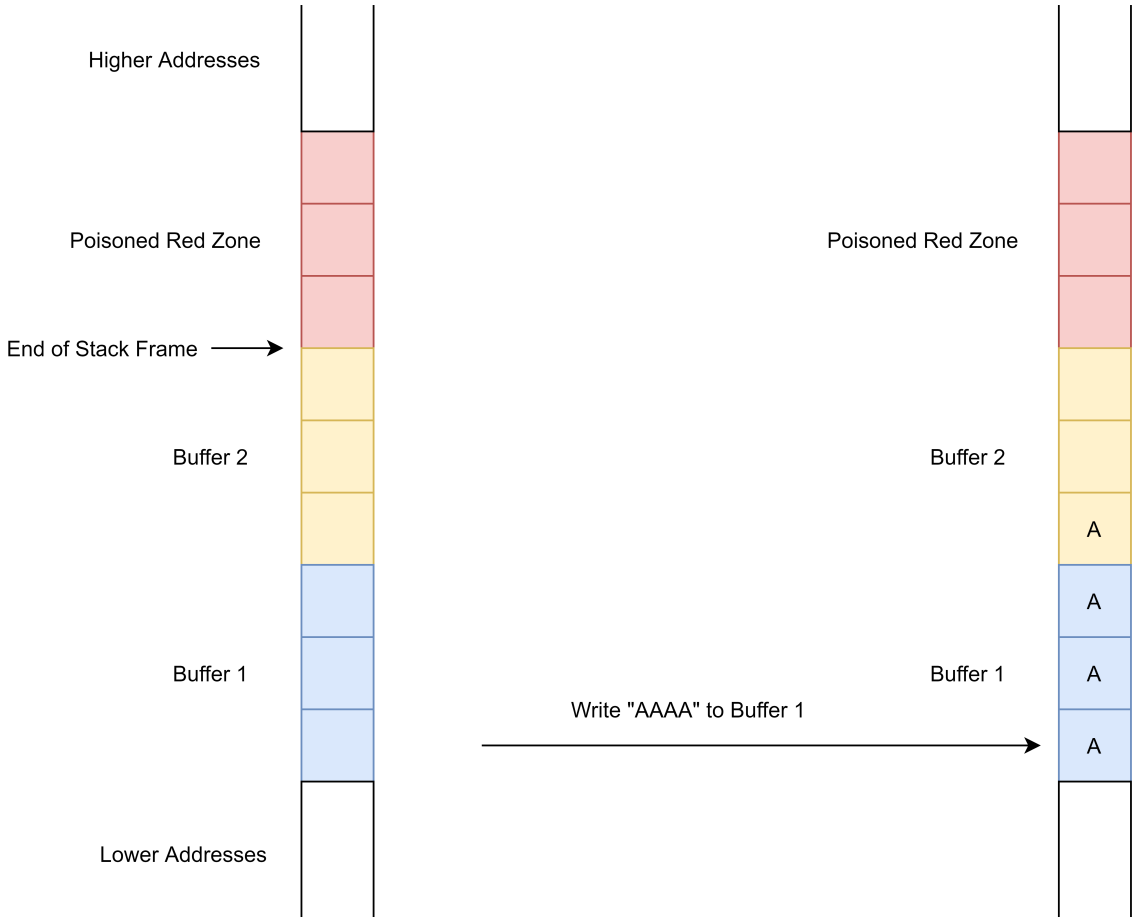

Figure 3.1: Buffer overflows buffer on the same stack frame

A possible workaround for the known conditions might be to add poisoned red zones only between stack frames. Individual stack frames can be identified, and experiments suggest that the avr-gcc compiler does not use offsets to address stack variables is other stack frames. This design has the disadvantage that it is not possible to detect when variables/buffers only overflow into other stack variables/buffers and when they do not overflow into the poisoned redzones. This is exemplified in figure 3.1. A buffer overflow occurs when the 4-byte long string "$AAAA$" is written to the 3-byte long *Buffer 1*. One

'A' is written to *Buffer 2*. This overflow is not detected because there is no poisoned red zone between *Buffer 1* and *Buffer 2*. Figure 3.1 does not show the stored return address for more clarity.

The design presented in the previous paragraph was not chosen. There are two reasons for this. First, there may be other unknown conditions. Second, it is unclear whether the avr-gcc compiler behaves according to the observations in the experiments in all cases.

Due to these unknowns, a design was chosen that does not change the memory addresses. The chosen design treats stack memory that stores return addresses as poisoned red zones. In the following, this chosen design is referred to as return address poisoning, or RAP for short.

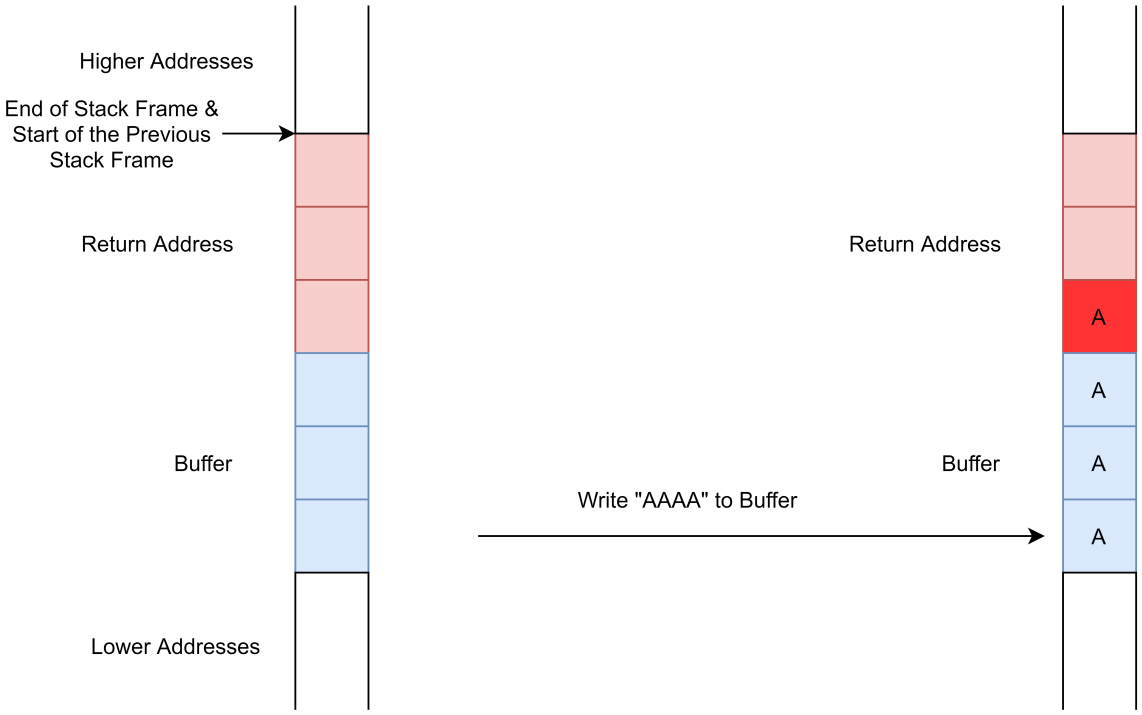

Figure 3.2: Buffer overflow write to poisoned red zone

Stack buffer overflows are detected when buffers overflow into memory that store a return address of a stack frame. It is assumed that a function is not allowed to overwrite its return address. Return addresses are pushed onto the stack when a function is called. This return address is thus located at the start of a stack frame. Furthermore, the stack grows downwards. Thus, return addresses in stack frames are located at higher addresses than the other data, such as buffers, in a stack frame of a function.

For this reason, a buffer overflow can overwrite a return address. Buffer overflows are detected in these cases. An example for such a case is shown in figure 3.2. Like the previous example 3.1, a 4-byte long string is written to a 3-byte long buffer. One character of the string overwrites a part of the return address. RAP detects this overflow.

A disadvantage compared to the previous design is that RAP cannot configure the size of the poison memory. The size of each poison memory block is the size of a return address.

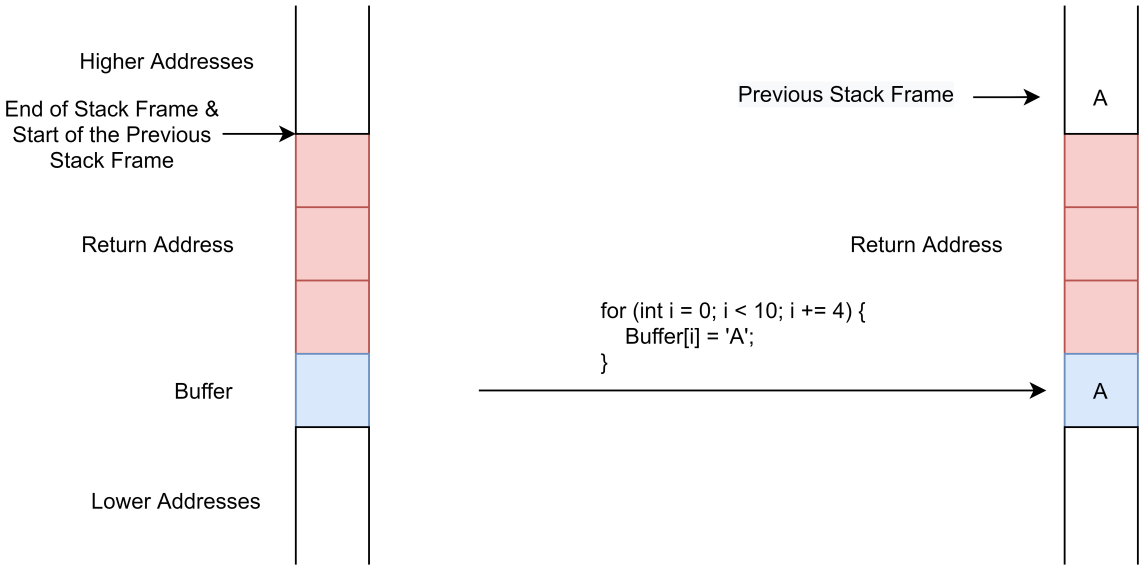

Figure 3.3: Buffer overflow write to every 4th Byte

Arduino microcontrollers with more than 128 KB in flash memory have an address size of 3 bytes, and 2 bytes otherwise [75]. Stack buffer overflows may be missed if, for example, only every 4th byte of a buffer if written to. This example is shown in figure 3.3.

One way to mitigate this disadvantage is to poison the saved base pointer in addition to the return address. The base pointer is also referred to as the frame pointer. Function prologues can push the base pointer onto the stack. Saved base pointers are stored on the stack after the return address. Poisoning both the base pointer and the return address doubles the size of the poisoned area.

This mitigation was not implemented, because storing the base pointer is in most cases omitted for optimization reasons. avr-gcc omits the base pointer when the '-fomit-frame-pointer' command line flag is used [76]. Enabling optimizations, i.e. when the '-O' command line flag is used, turns on the '-fomit-frame-pointer' flag [76]. This can be prevented by passing the '-fno-omit-frame-pointer' command line flag to avr-gcc [76]. However, a requirement that the user must modify and interact with the build systems of SUTs is not desirable. It adds an additional step before the user can start fuzzing. Furthermore, depending on the SUT build system and the user's experience with this build system, it can be difficult to add user supplied compiler flags.

A special case occurs when a function pops its own return address from the stack and new data is pushed onto the stack. This special case was observed with code written in assembly. The new data may be modified. Thus, the poisoned red zone is cleared in such cases.

3.2.2 Uninitialized Memory

A sanitizer for use of uninitialized memory (UUM) is implemented that reports a warning when an instruction whose operands and/or status flags must be initialized uses an

uninitialized operand and/or status flag. For example, a CPSE (Compare, skip if equal) instruction requires initialized operands. Another example: a BRCS (Branch if Carry Bit Set) instruction is usually preceded by a compare instruction that sets the carry bit. In this case, a BRCS instruction requires that the preceding compare instruction uses initialized operands. In contrast, a copy instruction does not require initialized operands [77].

A sanitizer for uninitialized memory must not report warnings when uninitialized operands or flags are used in 'safe' instructions [68]. For example, a struct may be copied even if not all members of the struct are initialized. However, a warning must be output when the program execution depends on an uninitialized value because this results in unpredictable and thus undefined behavior [77].

The chosen sanitizer for UUM design is based on Google's design for MemorySanitizer [68]. MemorySanitizer is a sanitizer for UUM that handles the requirements outlined in the previous two paragraphs correctly [68]. A major advantage of this design is that there are near-zero false positives [68]. MemorySanitizer is based on Memcheck [68]. Memcheck is a part of the Valgrind framework [78].

For each bit of program memory, MemorySanitizer stores the information whether this bit is initialized or not in a so-called shadow map [68]. Initialized memory, such as the static data in the .data section, is marked as initialized in the shadow map. Newly allocated memory is marked as uninitialized in the shadow map [68]. For example, new memory is allocated for a stack frame when a function is called. The shadow map is explained in more detail in the MemorySanitizer paper [68] section 3.1.

MemorySanitizer propagates information about initialized and uninitialized memory [68]. For example, storing an initialized value marks the destination memory location as 'initialized'. Similarly, storing an uninitialized value marks the destination memory location as 'uninitialized'. Interested readers find more information on this propagation in the MemorySanitizer paper [68] in section 3.2.

An operation can depend on more than one operand or flags [68]. In such cases, the memory location where the result is stored is marked as 'uninitialized' if at least one of the operands or flags are marked as 'uninitialized'.

UUM findings are difficult to debug [68]. An important step in debugging a UUM is to find out where the uninitialized value originated from. A value might undergo several copies and/or transformations before this value is used in an 'unsafe' instruction [68]. Between the origin, i.e. where the uninitialized value was first read, and the destination, i.e. the use of this value in an 'unsafe' statement, there may be a large number of instructions. Manually tracing back from destination to origin is time consuming. For this reason, this step is automated.

MemorySanitizer designed a solution for automated origin tracking [68]. Origin tracking is like propagation. However, unlike propagation of information about initialized and uninitialized memory, origin tracking propagates information about the origin [68]. This origin information includes the location in the source code that creates the uninitialized value [68]. Section 3.5 of the MemorySanitizer paper [68] provides a more detailed description of origin tracking.

So far, this section has summarized the design of MemorySanitizer. The rest of this

section will discuss the modifications that were designed for the use case of this paper, i.e. UUM sanitizer usage in a microcontroller emulator.

MemorySanitizer is implemented as a compiler optimization pass [68]. Therefore, MemorySanitizer operates with and instruments LLVM IR [68]. MemorySanitizer implements propagation and, when initialized operands are required, checks for initialized operands for all possible LLVM IR operations. In contrast, the implementation for the emulator operates on AVR machine code. Propagation and checks were implemented for the 124 AVR machine code instructions.

LLVM IR has more information about the source code compared to machine code. Thus, AddressSanitizer can output more information when a UUM is detected compared to the implemented version that only uses machine code information. This additional information includes for example the name of local variables where uninitialized values were stored [79].

When AddressSanitizer is used on a Linux host, an error can be reported when a system call argument is an uninitialized value [79] [68]. There are no system calls in Arduino software. Therefore, this feature is not implemented.

The registers must be included in the shadow map. These registers include for example the general-purpose registers, IO registers, and status registers. In contrast, MemorySanitizer, due to LLVM IR, operates with variables instead of general purpose and status registers [68]. The use of variables results in faster and simpler instrumentation in some cases [68]. For example, LLVM IR uses Boolean variables [68]. Machine code uses flags instead, and instructions often modify multiple flags, although often only one or no flag modification would be required [68].

In addition to these required changes, several changes were introduced to simplify the implementation and/or increase the performance of the emulator. MemorySanitizer operates on a bit level. The UUM sanitizer implementation operates on a byte level instead. This means that if one bit in a byte is marked as initialized, all 8 bits in that byte are marked as initialized. This change simplifies the implementation and increases the performance. Both are the case, for example, with the status flags. However, due to this change, UUMs may not be detected. For example, if a 1-byte variable is used as a bit field. Initializing one bit/flag in this field marks all bits/flags as initialized. Using one of the uninitialized flags in a 'unsafe' instruction is not detected in this case.

There are several special cases that the implementation handles. For example, the avr-gcc compiler uses the SUB subtraction instruction with two identical operands 'SUB X X', where X is a register, to set this register X to zero. In this case, register X must always be marked as initialized.

3.2.3 Timeout

An input may trigger an infinite loop due to a bug in the SUT. The fuzzer must detect such a case, so that the fuzzer does not get stuck executing one input endlessly. When a timeout occurs, the fuzzer must reset the emulator and execute the SUT with a new input.

For this reason, the user can specify a timeout. The timeout is specified in clock cycles. For example, a user can specify a timeout of one million cycles. If a single input takes the emulated microcontroller more than one million cycles to process, the input is reported as 'timed out'. After a timeout, the fuzzer resets and generates and executes a new input.

A timed-out input may or may not be considered a bug. For example, a project owner may specify that the SUT must process all possible input in less than X cycles. In another case, an input may be classified as timed-out because the user specified a timeout value that is too low.

An alternative to identifying timeouts via processed cycles is to use the system time of the host operating system that the fuzzer/emulator runs on. This system time approach is used by fuzzers like AFL and libFuzzer. However, the processed cycle approach was chosen because it is more accurate. In addition, the emulator already included a counter that keeps track of the number of processed cycles. The timeout sanitizer makes use of this built-in cycle counter. Therefore, the implementation effort and the processing overhead is small.

3.3 Fuzzer

A fuzzer is integrated into the emulator. Different types of fuzzers exist. The implemented fuzzer is graybox, mutation-based, and coverage-guided. This fuzzer design was chosen due to the good performance and ease of use of this type of fuzzer.

3.3.1 Fuzzer Type Alternatives

Instead of a graybox fuzzer, one could also use a whitebox fuzzer. Executing the SUT in an emulator works well for a whitebox fuzzer. Whitebox fuzzers require runtime information to generate new inputs. In this case, a whitebox fuzzer can collect this required runtime information, because the fuzzer has control over the emulation and access to all emulated data structures, systems, etc.

Even though there is the possibility to collect all runtime information, doing so and analyzing this information is slow. In practice, most whitebox fuzzers perform better than graybox fuzzers with small targets, but worse with medium to large targets. One reason for this is the path-explosion problem [80]. Path-explosion means that the number of execution paths double on every branch [80]. Most whitebox fuzzers have a runtime and/or memory complexity that is linear to the number of execution paths.

While microcontroller software is relatively small due to limited memory, most microcontroller software is too large, i.e. there are too many paths, for whitebox fuzzers to perform better than graybox fuzzer. For this reason, a graybox design is chosen.

One task of the fuzzer is to generate inputs. Mutation-based fuzzers generate new inputs by mutating previously generated inputs. A frequently used alternative input generation approach is the so-called generation-based input generation. Generation-based fuzzers use input templates or input specifications. Inputs are then generated based on this template.

When generation-based fuzzers are used, the user must specify the template. This template is typically created manually by the user. It can be difficult to create such a template, for example if the SUT expects complex inputs or if the user has little or no experience with the SUT and thus does not know the expected input structure. In the student exercises, the students likely do not know the expected input structure. For this reason, a generation-based fuzzer design was not chosen.

Compared to a blackbox fuzzer, a graybox approach enables the collection and analysis of lightweight runtime information. In this case, the lightweight runtime information is code coverage information. Coverage-guided fuzzers aim to test as much functionality of the SUT as possible. Code coverage information is used to select previously generated inputs that reach previously untested parts of the SUT. These selected inputs are used as the basis for newly generated inputs. The coverage information guides the input generation to deeper parts of the SUT [81]. Compared to generating a random input, as is the case in mutation-based backbox fuzzing, mutating an input with high code coverage increases the chance that the generated input reaches the same or higher code coverage. Thus, large portions of the SUT are tested, each time with a different input.

Over time, a coverage-guided fuzzer builds up a set of inputs that reach more and more code coverage. This is not the case with mutation-based blackbox fuzzers. To achieve good results with mutation-based blackbox fuzzers, it is often necessary that the user knows the expected input structure so that he can create seeds for the fuzzer. Students in the student exercises likely do not know the expected input structure.

Additionally, the requirement that the user must prepare a set of seeds to start fuzzing is one additional step before fuzzing can start. This is another disadvantage, because getting started with fuzzing in the student exercises should be as easy as possible.

A large set of seeds with good quality is beneficial in coverage-guided fuzzers too. It shortens the time until the fuzzer builds up a large set by itself. Furthermore, there may be paths in the SUT where it is unlikely that the fuzzer will reach the path in a timely manner. However, a large set of seeds is less required in coverage-guided fuzzers compared to mutation-based blackbox fuzzers.

The implemented fuzzer can start without seeds. A user can optionally specify user supplied seeds. This is done by passing the path to a folder to the emulator/fuzzer via command line arguments. The content of each file in this folder is one seed. A user can for example manually write seeds, download seeds from e.g. Github, or use the previous interesting inputs from a previous run as seeds for a new run.

A disadvantage of coverage-guided fuzzer is that collecting and analyzing code coverage information takes time. This time could have been spent executing the SUT with more inputs instead. However, despite this drawback, there is strong empirical evidence that coverage-guided graybox fuzzers on average outperform blackbox fuzzers [42].

For these reasons, a coverage-guided design was chosen. The design for collecting code coverage is explained later in section 3.4.

3.3.2 Mutator

The fuzzer's mutator is responsible for modifying an input in order to generate a new input. These modifications are called mutations. Different mutations exist. For example, a mutation can consist of flipping, adding, or removing a random bit of the input. Different mutators exist that make use of a different set of mutations.

The implemented fuzzer uses the mutator from libFuzzer. LibFuzzer is an open-source mutation-based fuzzer [43]. LibFuzzer is modular, so its mutator part can be reused in other fuzzers.

An alternative to reusing an existing mutator implementation is to implement a new mutator or to adjust an existing mutator implementation. LibFuzzer's mutator is designed to perform well on a wide variety of software targets. A custom mutator may be specialized to perform better with certain targets. For example, if the SUT expects human-readable input, a mutator can be modified so that the generated input consists exclusively of human-readable characters.

To allow the use of specialized mutators, a design is used that enables users to change mutators. The idea for this design is from AFL++ [21]. Mutators must be compiled as a shared object. The path to this shared object is passed to the emulator via a command line argument. During initialization, the emulator dynamically loads the shared object. Mutators must implement two functions: mutator_init and mutator_mutate. mutator_init is called once during emulator initialization. mutator_mutate is called repeatedly to generate inputs. The input that should be mutated is passed to the mutator_mutate function via a function argument.

3.4 Coverage

Graybox mutation-based fuzzers must determine whether a generated input is interesting. Interesting inputs are added to the set of previous interesting inputs. Previous interesting inputs are used by the mutator as basis for newly generated inputs. Coverage-guided fuzzers make this classification based on the coverage reached by the input.

There exist different types of coverage metrics. These include for example basic block coverage and edge coverage. Edge coverage is also referred to as branch coverage or transition coverage [81]. Edge coverage is popular in coverage-guided fuzzers. For example, AFL, AFL++, and libFuzzer use edge coverage or a coverage type that is based on edge coverage [43] [82]. Edge coverage provides more information compared to basic block coverage. Interested readers find more information on edge coverage on this [83] website.

The emulator captures branch coverage by instrumenting all instructions that can jump to different destination addresses. These are conditional jump instructions, as well as calls and jumps to a destination address that is not constant (e.g. jump to the value in register XYZ). That means the instruction that is executed after such an instruction can differ [84]. In the following, these instructions are referred to as branch instructions. Machine code uses branch instructions to implement e.g. if-statements and switch statements. An example AVR branch instruction is CPSE (Compare, skip if equal) [52].

There is no need to instrument instructions that are not branch instructions. An example sequential flow instruction is 'add' or 'nop' (no operation) [84]. After a sequential flow instruction, the instruction that follows this sequential flow instruction is always executed [84].

All branch instructions are instrumented. At runtime, this instrumentation checks whether the edge from source to destination has been executed previously. The source is the address of the instruction currently being executed, i.e. the program counter. The destination is the target instruction address of the jump.

If a source and destination pair is encountered for the first time, the input that triggered this pair has increased the branch coverage and therefore this input is added to the set of previous interesting inputs. This coverage tracking does not require a large runtime overhead, even for larger SUTs. Triggered source and destination pairs are stored in a hash set. Hash set lookups are performed in constant time.

In addition to branch instructions, jump and call instructions with a constant destination address are instrumented too. This is not required for identifying interesting inputs. It is only required for the Coverage Explorer. The Coverage Explorer is part of the User UI, which is explained later in section 3.9.

3.5 Identify interesting Program States via Data Values

Coverage-guided fuzzers identify whether an input increases the code coverage. These inputs are marked as 'interesting'. Generated inputs are based on previous interesting inputs. Although this input generation approach based on coverage feedback is often very effective, there are programs that use for example complex state machines where this coverage-guided approach fails [85]. Finite state machines are a common design pattern in embedded systems [86].

To aid the fuzzer in such cases, Aschermann *et al.* have proposed an annotation mechanism [85]. This mechanism allows the user to specify the location of data that represents the internal state of the SUT [85]. If an input results in an interesting value for the user-specified data, this input is marked as 'interesting' and used for the generation of future inputs. For example, Aschermann *et al.* use the position coordinates of Mario in the Super Mario Bros game as the data that represents the internal state of the program [85]. This enabled the fuzzer to defeat Bowser in Super Mario Bros and the annotation mechanism was also used to discover several security vulnerabilities [85].

Note that this mechanism extends the coverage-guided approach, and it does not replace it. This means that both inputs that increase coverage and inputs that lead to interesting values are used to generate future inputs.

In addition to the location of data, the user must specify when the data at this location is interesting [85]. For example, data can be interesting if it has not been observed before or the data can be interpreted as an unsigned integer value and such a value can be interesting if it is the largest observed so far [85]. For example, Aschermann *et al.* marked an input as interesting if Mario's x-coordinate is the largest observed so far [85].

The chosen design to specify and identify interesting program states via data values is lika Aschermann's annotation mechanism. However, they differ due to different conditions. For example, in Aschermann design the annotations are specified in the SUT's source code [85]. Aschermann's implementation extends the AFL fuzzer [85]. Instrumentation is added via a Clang compiler optimization pass, which notifies AFL when a interesting value is observed [85]. In contrast, because a compiler optimization pass is not used and because the fuzzer runs outside of the emulation, in our design the user specifies the annotations via the User C API and the implementations differ. Both approaches achieve a similar result, but in different ways.

The C API provides the add_state and create_state_patch functions. These two functions are used together with the patch_instruction function from the user API. The following listing 3.11 shows a typical usage of these functions:

```
1 patch_instruction(get_symbol_address("some_function", avr),
2                   add_state,
3                   create_state_patch("some_variable", UNIQUE, avr));
```

Listing 3.11: add_state example

In this example, *some_function* is a function from the SUT and *some_variable* is a global variable in the SUT. *UNIQUE*, the second argument for the create_state_patch function, specifies that if the value in *some_variable* has not been observed so far, the current input is marked as interesting.

Instead of *UNIQUE*, the implementation also supports MAX and MIN. With MAX, the current input is marked as interesting if the value in *some_variable* is the largest observed so far. Similarly with MIN, the current input is marked as interesting if the value is the smallest observed so far.

In the example listing 3.11, these checks are performed when the SUT calls *some_function*. A user can check different variables, can check the same variable at different times, i.e. at different SUT addresses, and can check for both new MINs and new MAXs of the same variable at the same time.

The implementation supports data values with a maximum size of 64 bits. In addition, the values are always interpreted as an unsigned integer in the MIN and MAX checks. These constraints simplify the implementation. Future work could extend the implementation to support data values of arbitrary sizes and other data types, such as floats.

3.6 Crash Handler

A fuzzer might trigger the same bug multiple times. In the following, inputs that trigger a bug are referred to as crashing inputs. Providing all crashing inputs to the user is confusing. The user is more interested in how many different bugs were found. Furthermore, users should be provided with an input for each different bug. The user can use this input to reproduce the error.

To achieve this, crashing inputs must be deduplicated. This process can be automated. Several techniques have been developed to deduplicate crashing inputs [87]. For example,

a crashing input can be deduplicated based on the program counter, the coverage, or the stack trace of when the sanitizer detects the crash [87]. Program counter-based deduplication classifies a crashing input as 'unique' if no previous crashing input had the same crash address. The crash address is the program counter at the time when the sanitizer detects the crash. Interested readers find more information on coverage-based and stack trace-based deduplication in the paper [87] in section 7.2 and section 7.3, respectively.

All these methods can misclassify crashing inputs [87]. Deduplication based on the program counter has the lowest chance of misclassifying an input as 'unique', but it has the highest chance of misclassifying an input as duplicate. In other words, program counter-based deduplication outputs the least number of crashing inputs at the expense of missing 'unique' crashing inputs. The first crashing input is not missed.

The implemented crash handler uses the program counter method to deduplicate crashing inputs. This method was chosen because in a learning environment, such as the student exercises, a smaller number of crashing inputs is less overwhelming.

The crash handler passes information about 'unique' crashing inputs to the user UI user. The user UI is explained later in section 3.9. The information about the crashing input depends on what type of bug is detected. The crashing address is always passed to the user UI server. Additionally, if the crashing input is due to a UUM, origin tracking is used to pass the origin address as well.

3.7 Emulator Reset

After the SUT has processed an input or if a non-recoverable error is detected, the emulator must restart/reset the SUT process so that the SUT can be tested with another input. Existing fuzzers solve this problem in different ways.

LibFuzzer repeatedly calls a function in an endless loop [43]. In each loop iteration, a new input is passed to this function [43]. LibFuzzer does not restart or reset the SUT [43]. One advantage of this approach is that it is performant.

When working with libFuzzer, a user must consider that an SUT can have global state. A major disadvantage of libFuzzer's approach is that the SUT or the user that tests the SUT with libFuzzer must implement logic that resets any relevant state. Relevant state includes for example global variables, the state of the interrupt system, of the timer system, or of the I/O ports. This state can change the program's behavior and it can make individual test runs nondeterministic. If a bug is found, but the tests are nondeterministic, there is a high probability that the user will not be able to reproduce the bug. For this reason, relevant state must be reset after every input.

Implementing logic to reset relevant state is an additional step before a user can start fuzzing. Additionally, if a user is unfamiliar with a SUT, it is difficult to identify relevant state. The students that participate in the student exercises are likely not familiar with the SUTs. For these reasons, libFuzzer's reset design was not chosen.

The implemented reset mechanism reuses the implementation of the 'System Reset' mode from the watchdog timer. The watchdog timer can be used as a safety feature [88]. When

enabled, a timer runs independently of the system [88]. The timer can be reset [88]. When the timer runs out, an action occurs. The action depends on the operation mode of the watchdog timer [88]. AVR provides the following three operation modes [88]:

- Interrupt mode
- System Reset
- Interrupt and System Reset

Not every AVR microcontroller supports a watchdog timer. Furthermore, not every microcontroller supports all three operation modes [88]. However, the microcontroller that is used in this project, namely the Arduino Mega 2560, supports the watchdog timer and the System Reset mode [89]. simavr supports the watchdog timer when the Arduino Mega 2560 is emulated [5].

When the watchdog timer runs out in System Reset mode, the watchdog timer forces a reset [88]. This reset restarts the microcontroller.

A major advantage of this design is that the watchdog timer implements the logic to reset all relevant state. If libFuzzer's design would be used, this logic had to be implemented by the user.

One disadvantage of the chosen approach is that it is less performant compared to libFuzzer's reset design. A system reset resets non-relevant state, which takes time. Furthermore, potential time-consuming initialization steps, such as reading a large configuration file, are performed on every input [90]. Initialization steps that are performed before the SUT processes an input from the fuzzer could be executed only once. A more performant design could create a snapshot before the SUT processes an input and after the initialization steps. QEMU provides such a snapshot feature [91]. simavr does not provide a snapshot feature.

AFL++ provides a 'Deferred Initialization' feature [90]. This feature is lika a snapshot. AFL++ uses the clone Linux system call after the SUT has performed its initialization steps [90]. By default, clone is called before the SUT executes its main function [90]. Users can change the default behavior [90]. Each cloned process receives one input from the fuzzer. A cloned process exits after it has processed the input.

The design of AFL++ was not chosen, because a clone system call is not always more performant compared to the chosen design. This depends on the time that the SUT initialization steps take. In addition, the design of AFL++'s complicates the fuzzer/emulator implementation because the parent AFL++ process must use inter process communication (IPC) to communicate with the cloned child processes and vice versa [15]. This IPC also reduces the fuzzer's performance [15]. For example, AFL++ uses IPC to pass the input from the parent process to a cloned process and cloned processes pass coverage information to the parent process.

3.8 When to Reset

This section discusses the options for when to reset the SUT process and restart it with a different input. These options can be divided into options that are automatically enabled and options that can be optionally enabled by the user.

The SUT is always reset when the sanitizer discovers a non-recoverable error. Non-recoverable errors include for example a buffer overflow or when the SUT tries to execute a non-existent instruction. Additionally, when the user sets a timeout, the SUT is reset when a timeout occurs.

A user has several options to configure and specify when the SUT should reset. The first option is via the user API. The user API provides the 'fuzz_reset' function. This function can be used by the user in combination with the 'patch_instruction' function to reset when the SUT is about to execute an instruction at a specified address. In addition, a user can specify a timeout. Timeouts were discussed in section 3.2.3. Users can also specify whether the user UI server should be notified when a timeout is found.

3.9 User UI

The primary purpose of the user UI is to provide information to the user about the status of the fuzzing run. This information includes the coverage reached by the fuzzer and the average number of processed inputs in the last 10 seconds. In addition, the user UI stores inputs that crash the SUT so that the user can reproduce crashes. The user UI is updated in real time, i.e. while the fuzzer is fuzzing.

The user UI is a separate process. In the following, this process is referred to as the UI server. The main reason why the UI server is a separate process is because the emulator and the fuzzer are implemented in C and the UI server is implemented in Python. Python is more suitable for the UI use case, because it allows easier integration of third-party tools, such as plotting libraries and command line-based coverage tools.

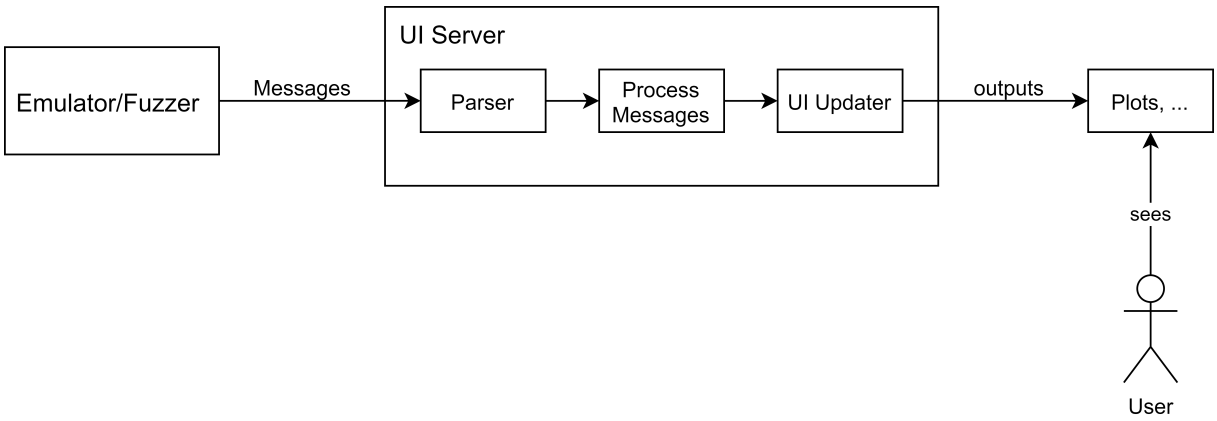

Figure 3.4: UI Server Overview

Figure 3.4 shows the inputs and outputs, as well as the main components of the UI server. The emulator/fuzzer process sends messages to the UI server via sockets. A message consists of a header and a body part. The header has a fixed length and includes two fields: an ID that specifies the type of the message and the length of the body. The body has a variable length.

The Parser component reads raw bytes from a socket. After all bytes of a message have arrived, the message body is parsed and deserialized. Deserialized data is then passed to

a function of the Process Messages component via function arguments. The function that is called depends on the type of the message in the message header.

The Process Messages component maintains data structures. These data structures are used by the UI Updater. The UI Updater mainly consists of third-party tools, such as plotting libraries. The input for these third-party tools is data that is stored in the data structures of Process Messages.

To enable this, the primary task of Process Messages is to update the data structures with data from parsed messages. Process Messages translates messages into the format that the third-party tools expect.

There are four types of messages:

- Initial SUT Information
- New Coverage
- New Crash
- Updated Fuzzer Statistics

The first message that is sent is the 'Initial SUT Information' message. This message is sent once during emulator/fuzzer initialization. The message contains the path to the SUT executable file that is emulated. This file is later used in combination with the addr2line tool by Process Messages to convert instruction addresses into file names and line numbers [92]. The usage of this is explained later.

A 'New Coverage' message is sent each time an edge is executed for the first time. The message body includes the source and destination addresses of the executed edge. In addition, the message body contains the input that executed this edge. This input is a previous interesting input. Previous interesting inputs are stored to disk as part of the message processing step in the Process Messages component. The main purpose of storing these inputs to disk is that they can be used as seeds in a future fuzzing run.

The UI Updater uses this coverage data for three representations: in text, as an edge coverage over time graph, and in HTML pages that contain the SUT source code that is annotated with coverage information. UI Updater periodically writes the number of edges found to a file. This file is called fuzzer_stats. An example fuzzer_stats file content is shown in listing 3.12. The fourth line in this listing starts with *edges_found* and specifies that the fuzzer has encountered *564* unique edges in its current run. fuzzer_stats also contains information from messages of the 'New Crash' and 'Updated Fuzzer Statistics' types.

```
1  Date: 2021-04-25 11:40:07
2  average_inputs_executed_last_10_seconds: 458.7431663269034
3  bad_jump_count: 0
4  edges_found: 564
5  fuzzer_start_time: Sun, 25 Apr 2021 11:38:48 PDT
6  inputs_executed: 20000
7  invalid_write_address_count: 0
8  max_depth: 7
9  previous_interesting_inputs: 59
10 reading_past_end_of_flash_count: 0
11 stack_buffer_overfow_count: 0
```

```
stats_update_time: Sun, 25 Apr 2021 11:39:32 PDT
timeout_count: 0
total_crashes: 0
uninitialized_value_used_count: 0
```

Listing 3.12: fuzzer_stats Example

Coverage data is also used to generate an edge coverage over time graph. An example graph is shown in figure 3.5. The x-axis is in logarithmic scale. UI Server uses the third-party tools matplotlib and seaborn to generate the graph [93] [94].

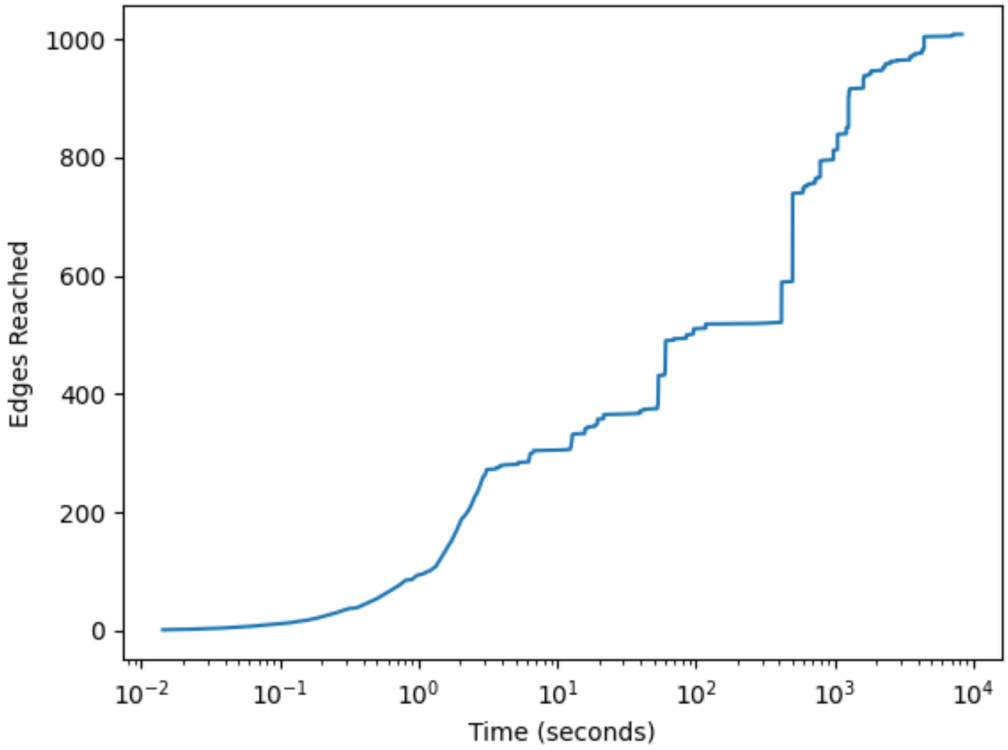

Figure 3.5: Coverage over Time Graph Example

The third use case for the coverage data is with the so-called Coverage Explorer. The Coverage Explorer consists of HTML pages. These HTML pages show the SUT source code. Lines that were executed have a green background. An example page of the Coverage Explorer is shown in figure 3.6.

The Coverage Explorer is implemented with the third-party tools gcovr [95] and the disassembly library Amoco [96]. As input, gcovr requires source code locations. A source code location is the path to a source code file and a line number. The Process Messages component uses addr2line to convert source and destination addresses of executed edges to source code locations [92]. UI Updater passes these source code locations via files to gcovr and executes gcovr periodically.

Messages of the 'New Crash' type are sent when the emulator found a new 'unique' crash. Section 3.6 discusses when a crash is classified as 'unique'.

The message body for 'New Crash' messages consists of the following:

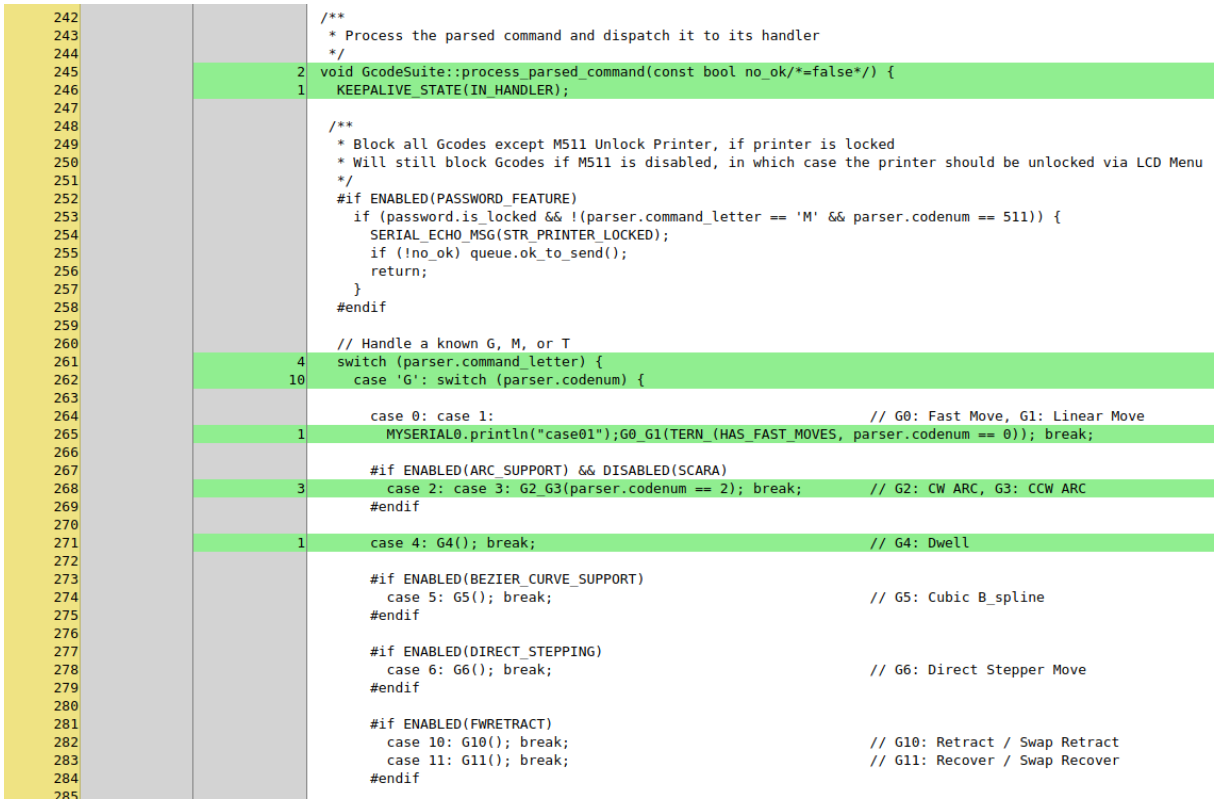

Figure 3.6: Coverage Explorer Example HTML Page

- The type of the crash (Buffer Overflow, UUM, etc.).
- The crashing address, i.e. the address of the instruction that was executed when the sanitizer or emulator detected the crash.
- The crashing input, i.e. the input that caused the crash.
- The stack trace at the time when the sanitizer or emulator detected the crash.
- Additionally, if the type of crash is UUM, the origin address is included as well.

A crashing input is written to a file. The name of this file includes the type of the crash and the crashing address. If the type of crash is UUM, the filename includes the origin address as well. An example filename for a UUM crash is 'uninitialized_value_used_at_109e2 _with_origin_c8e0'. In this example, the crashing address is 0x109e2. The origin address is 0xc8e0.

Stack traces are processed before they are stored to disk. The stack trace in messages consists of addresses. Each of these addresses represents one stack frame of the stack trace. simavr has implemented the logic to collect these stack trace addresses [51]. The Process Messages component uses addr2line to convert these addresses to source code locations [92]. Both the addresses and the source code locations of the stack trace are written to a file. An example file with a stack trace is shown in the following listing 3.13:

```
Stackframes at crash:
    #0 0x1a20 in __do_global_ctors at ??:?
    #1 0xfb1e in main at /arduino/hardware/avr/1.8.3/cores/arduino/main.cpp:43
```

```
4    #2 0xf376 in _ZN10GcodeSuite3G28Ev at /src/gcode/calibrate/G28.cpp:319
```

Listing 3.13: Stack Trace Example

addr2line uses debug information to convert addresses to source code locations [92]. Debug information is stored in the executable file. An executable file may not contain debug information for all its addresses. In such cases, the source code location cannot be determined. This case is indicated by '??:?' in stack trace files.

The fuzzer_stats file is also updated whenever a 'New Crash' message arrives. This file is shown in listing 3.12. For example, if the type of the crash is UUM, the *uninitialized_value_used_count* counter is incremented.

Messages of the 'Updated Fuzzer Statistics' update the fuzzer_stats file as well. 'Updated Fuzzer Statistics' messages contain the following:

- inputs_executed: How many inputs have been executed.
- total_crashes: The total number of unique and non-unique crashes.
- max_depth: The number of stack frames in the largest stack trace observed so far.

Process Messages also calculates the average_inputs_executed_last_10_seconds metric. 'Updated Fuzzer Statistics' messages are sent repeatedly.

4 Implementation

This section describes relevant parts of the implementation.

4.1 User API

The User API implementation consists of three parts:

- Data structures that store when and what functions should be executed. The users configures the 'when' and 'what' via the patch_instruction function calls.
- Prior to emulating an instruction, checking whether user-specified functions should be called.
- Helper functions (i.e. abstractions) for working with the emulator and fuzzer.

4.1.1 Patch Instruction

Patched instructions are stored in a dictionary called patches. The keys for the patches dictionary are instruction addresses. Values are lists of structs. These structs are called patch. The Patch struct consist of two members: a function pointer and a void pointer. The implementation uses a third-party library for the list and dictionary data structures from [97].

```
1 int patch_instruction(avr_flashaddr_t vaddr,
2                        void *function_pointer,
3                        void *arg);
```

Listing 4.1: patch_instruction Function Prototype

If *patch_instruction* (listing 4.1) is called, a Patch struct is appended to the list at key *vaddr* of the patches dictionary. Patch struct members are initialized with the *patch_instruction* parameters *function_pointer* and *arg*.

Prior to emulating an instruction, the emulator checks whether the patches dictionary contains a key with the current instruction address, i.e. the program counter. If this is the case, the emulator iterates over all structs in the list and calls the function pointed to by *function_pointer* with the *arg* as function argument.

4.1.2 Direct Memory Access

The emulator stores the RAM of the SUT in a block of memory. Users call the helper function write_to_ram to write to the SUT's RAM. A pointer to the start of the RAM block is stored in the avr struct. The avr struct calls this pointer 'data'.

The *write_to_ram* function implementation (listing 4.2) copies user supplied data to the SUT's RAM, i.e. into the 'data' memory block. In addition, the shadow map must be updated. Shadow map is part of the UUM sanitizer. The written bytes are marked as 'initialized' via the *set_shadow_map* function. Otherwise UUM will output false positives. The shadow map implementation is later explained in section 4.2.2.

```
1 void write_to_ram(uint32_t dst, void *src, size_t num_bytes,
2                           avr_t *avr) {
3   memcpy(avr->data + dst, src, num_bytes);
4
5   set_shadow_map(dst, num_bytes, 1, avr);
6 }
```

Listing 4.2: write_to_ram Function

4.1.3 Symbols

The simavr implementation has already implemented logic to read and parse all symbols from an executable file in ELF format [98]. simavr uses libelf for this purpose [98] [99]. libelf is part of elfutils [99].

```
1 uint32_t get_symbol_address(char *symbol_name,
2                              avr_t *avr);
```

Listing 4.3: get_symbol_address Function Prototype

Simavr's symbol reading and parsing logic is reused for the symbol name to address translation. This logic was extended for this project. Namely, read symbols are added to a dictionary called symbols. Keys are strings that store the symbol name. Values are integers that store the symbol address. A pointer to the symbols dictionary is stored in the *avr* struct. The *get_symbol_address* function (listing 4.3) looks up and returns the address value for the given *symbol_name* key via the symbols dictionary.

AVR uses a Harvard architecture [100]. In contrast to the von Neumann architecture, the Harvard architecture has separate address spaces for code and for data [101]. Thus, the same address can be used to access two different values: one code value and one data value.

The ELF format has no mechanism to associate one address with two values. For this reason, ELF symbol addresses for data values have a constant offset of 0x800000 added to the actual address. The *get_symbol_address* function implementation takes this into account. If *get_symbol_address* detects an offset, the offset is subtracted from the value before the value is returned.

4.1.4 External Interrupts

To emulate the effects of an external interrupt, simavr calls the *avr_raise_interrupt* function. This *avr_raise_interrupt* function was already implemented. Via which GPIO

pin the interrupt is received is specified via a function argument and encoded in an avr_int_vector_t struct.

The C API helper function for external interrupts differs in how the interrupt is specified by the user. Instead of specifying a avr_int_vector_t struct, the user specifies the interrupt via a pin number of the Arduino Mega2560 board.

C API helper function *raise_external_interrupt* (listing 4.4) converts the *pin* number to the corresponding avr_int_vector_t struct. The avr->interrupts.vector array stores avr_int_vector_t structs for all pins that are usable for interrupts with the Arduino Mega2560 board. The *avr_raise_interrupt* function is called with the avr_int_vector_t struct for the pin with pin number *pin*.

```
1  void raise_external_interrupt(uint8_t pin, avr_t *avr) {
2    avr_raise_interrupt(avr, digitalPinToInterrupt(pin, avr));
3  }
4
5  avr_int_vector_t *digitalPinToInterrupt(uint8_t pin, avr_t *avr) {
6    uint8_t vector_index = 0;
7    switch (pin) {
8    case 2:
9      vector_index = 6;
10     break;
11 ... more cases for the other digital pins that are usable for interrupts with
       the Arduino Mega2560 board.
12
13   return avr->interrupts.vector[vector_index];
14 }
```

Listing 4.4: raise_external_interrupt Implementation

4.2 Sanitizer

This section describes the implementation of the three sanitizers that were implemented and integrated into simavr.

4.2.1 Stack Buffer Overflow

A stack buffer overflow is detected when the SUT overwrites the poisoned return address. The sanitizer must keep track of the addresses of the return address in all stack frames. For this purpose, all instructions that push a return address on the stack and instructions that pop a return address from the stack must be instrumented. These instructions include the CALL instructions and various variants of the CALL instruction, and the RETURN instruction [52].

Simavr has already implemented logic to record call and return instructions. To do this, simavr maintains a Last In First Out stack called *stack_frame* [102]. When a call instruction is emulated, the current program counter is pushed onto *stack_frame* [51].

When a return instruction is emulated, a program counter is popped from *stack_frame* [51]. The stack buffer overflow sanitizer uses this to know at which addresses the return addresses are stored.

Simavr calls the *avr_core_watch_write* function whenever the SUT writes to its RAM. Relevant parts of this function are shown in listing 4.5. The *addr* argument is the target of the write operation. Return addresses are 3 bytes long.

```
void avr_core_watch_write(avr_t *avr, uint16_t addr, uint8_t v) {

... Sanity Checks

  int frame_index = avr->trace_data->stack_frame_index - 1;
  for (int i = frame_index; i >= 0; i--) {
    // We add +1 here because .sp points to 1 below where the return
    // address is stored.
    uint16_t stack_return_address = avr->trace_data->stack_frame[i].sp + 1;
    if (stack_return_address <= addr &&
        addr < stack_return_address + avr->address_size) {
      stack_buffer_overflow_found(avr, avr->pc);
    }
  }
...

  avr->data[addr] = v;
}
```

Listing 4.5: avr_core_watch_write Function [51]

The *avr_core_watch_write* function is extended with instrumentation that checks whether *addr* points to a return address. If this is the case, the Crash Handler is notified via a call to *stack_buffer_overflow_found*, and the emulator is reset.

In the current implementation, the addresses of return addresses are stored in a stack (i.e. an array). The checks whether *addr* points to a return address does not scale well with the number of stack frames. For a more performant solution, a hash set for fast lookups could be used in addition to the stack. The stack/list is still needed to know which element should be removed from the stack and the hash set when the emulator emulates a RETURN instruction.

4.2.2 Uninitialized Memory

The UUM sanitizer implementation can be structured into the following four parts:

- Initialization of the shadow map and the origin tracking map.
- Propagtion of information about initialized and uninitialized bytes.
- Detection of UUM in instructions that require initialized memory or flags.
- Reporting of UUM findings.

Initialization

During emulator initialization, the UUM sanitizer allocates two blocks of memory. Each memory block has a size of 8 kilobytes. The Arduino Mega 2560 has 8 kilobytes of RAM [26].

In listing 4.6, a pointer to the start of the memory block for the shadow map is stored in the *shadow* variable. This variable is a member of the *avr* struct. Similarly, avr's member *shadow_propagation* points to the memory block of the origin tracking map. Each map is a one-to-one mapping from bytes in the map to bytes in the SUT's RAM. For example, the byte at index 0 in the shadow map specifies whether the address 0 in the SUT's RAM is defined or undefined. The shadow map stores a '1' if the byte at this index is defined, and '0' if this byte is undefined.

```
1  void initialize_uninitialized_sanitizer(avr_t *avr) {
2    avr->shadow = calloc(1, 1 << 16);
3    avr->shadow_propagation = calloc(sizeof(uint32_t), 1 << 16);
4
5    for (int i = 0; i < 0x200; i++) {
6      avr->shadow[i] = 1;
7    }
8  ...
9    uint32_t __data_start = get_symbol_address("__data_start", avr);
10   uint32_t __data_end = get_symbol_address("__data_end", avr);
11 ...
12   for (uint32_t i = __data_start; i < __data_end; i++) {
13     avr->shadow[i] = 1;
14   }
15 ...
16 }
```

Listing 4.6: initialize_uninitialized_sanitizer Function

Bytes in the .data section, as well as the registers such as the stack pointer and IO registers are initialized when the SUT starts. Those bytes are marked as 'defined' in the shadow map during emulator initialization. Line 5 to 7 in listing 4.6 set all registers and IO Registers to 'defined. The Registers and IO Registers have address 0 to 0x200.

The start and end address of the .data section is identified via the symbols *___data_start* and *___data_end*. Error checks are not shown in listing 4.6 for more clarity. For example, the UUM sanitizer implementation takes into account that these symbols may not exist. They could be removed from the ELF executable, or the ELF executable might not have a .data section.

Propagation

Propagation is implemented by instrumenting the instructions that work with data in the SUT's RAM, i.e. registers, IO registers, stack, various sections etc. For example, the 'nop' (no operation) instruction does not work with data and required no instrumentation. Most instructions work with data and were therefore instrumented.

```
1  #define sprop_2(a, b, c) ((a) ? (a) : ((b) ? (b) : ((c) ? avr->pc : 0)))
2  ...
3  // Read from flash at address 'program counter'.
4  uint32_t opcode = _avr_flash_read16le(avr, avr->pc);
5  ...
6  switch (opcode) {
7  ...
8    case 0x1400: { // CP -- Compare
9      get_vd5_vr5(opcode);
10     uint8_t res = vd - vr;
11     _avr_flags_sub_zns(avr, res, vd, vr);
12     avr->shadow[SF] = avr->shadow[d] & avr->shadow[r];
13     avr->shadow_propagation[SF] = sprop_2(avr->shadow_propagation[d],
14                                           avr->shadow_propagation[r],
15                                           !avr->shadow[SF]);
16   } break;
17 ...
```

Listing 4.7: compare Instruction Emulation [51]

Listing 4.7 shows the code that is executed to emulate the Compare instruction. The *get_vd5_vr5* is a macro which initializes the variables d and r. d and r store the two operands of the compare instruction. These operands are addresses.

Line 12 in listing 4.7 implements the propagation for the compare instruction. The compare instruction sets status flags. The *SF* macro contains the address of the status flags register. If both operands of the compare instruction are defined, the shadow map at index *SF* is set to 1, i.e. defined. If either operand is undefined, the shadow map at index *SF* is set to 0, i.e. undefined. The shadow map at index d and r specifies whether address d or r are defined or undefined.

The origin propagation instrumentation for the compare instruction is shown in listing 4.7 in line 13 to 15. This instrumentation uses the macro that is defined in line 1. The instrumentation handles three possible cases:

- Both operands are initialized. The origin value does not matter when both operands are defined. In this case, the origin propagation map *shadow_propagation* at index *SF*, d, and r is zero.
- An origin value is propagated because a UUM has occurred already. For example, an uninitialized value was copied to address d. In this case, the *shadow_propagation* at index d and/or r store an origin value that is not zero. One non-zero (i.e. undefined) origin value is propagated and stored at index *SF* in the shadow map.
- It is the first time that an UUM is noticed. This is the case if the *shadow_propagation* at index d and index r store the value zero. In this case, the origin value is set to the address of the current instruction, i.e. the program counter. This origin value is stored at index *SF* in the shadow map.

Some instructions have one or three operands. Furthermore, some instructions can modify multiple values, e.g. the status flags register and another register. The implementation handles these cases similarly to the compare instruction example.

There are also several special cases, such as 'SUB X X' that was explained in the design section 3.2.2. The implementation handles these special cases as well. For example, in the 'SUB X X' case, the X register is always set to defined in the shadow map.

Detection and Reporting

Instructions that require initialized operands and/or flags check whether their operands and/or flags are initialized via the shadow map. If they are not initialized, the Crash Handler is notified and the emulator resets. This notification message includes the crashing address and the origin address. The origin address is read via the origin propagation map.

4.2.3 Timeout

Simavr has already implemented a counter that keeps track of how many clock cycles the emulated microcontroller has been running for [51]. Prior to emulating an instruction, this counter is used to check whether more than the specified 'timeout' number of clock cycles has passed since the emulator started or reset. If more cycles have passed, the Crash Handler is notified and the emulator resets.

4.3 Fuzzer

The fuzzer maintains a dynamic array where each element is one previous interesting input. If the user specified seeds, this list is initialized with the user-supplied seeds. Otherwise, the list is initialized with one input, namely the string 'A'.

The dynamic array data structure is from the third-party Collections-C library [103]. Previous interesting input elements in this array are 'Input' structs. An Input struct has two members: a memory block that stores the input, and an integer that stores the length of this input.

When seeds are supplied, the fuzzer creates one Input struct for each file in the seeds folder. Each Input struct is initialized with the content of one file in the seeds folder. These input structs are added to the previous interesting inputs array.

During emulator initialization and after every reset of the emulator, the fuzzer randomly selects one Input struct from the previous interesting inputs array. The input bytes and the input length are copied to another input struct that is called 'current_input'. The current_input struct is passed to the mutator. The mutator modifies the input. The previous interesting input is copied so that the selected previous interesting input is preserved for future input generations. Mutations can change the size of the input. The length of the input is kept up to date via the current_input struct.

Users have access to the current_input struct via the avr struct. Helper functions from the user API, such as the *write_fuzz_input_global* function described in section 3.1.2, use the current_input struct as well.

When an input executes one or multiple previously unseen edges, the current_input is duplicated, and the copy is added to the previous interesting inputs array.

The fuzzer reuses the mutator from libFuzzer. This idea of reusing mutators from other fuzzers is also used by AFL++. AFL++ calls this the Custom Mutator API [19]. The implementation uses the same idea as AFL++ [104]. The fuzzer is compiled as a shared object. At runtime, the emulator loads this shared object into memory via the dlopen function [105] and calls functions from the mutator.

4.4 Coverage

Collecting coverage is implemented by instrumenting all branch instructions. An example branch instruction is 'Compare, skip if equal' (CPSE). The relevant code that emulates the CPSE instruction is shown in listing 4.8.

```
// Read from flash at address 'program counter'.
uint32_t opcode = _avr_flash_read16le(avr, avr->pc);
...
switch (opcode) {
...
  case 0x1000: { // CPSE -- Compare, skip if equal
    get_vd5_vr5(opcode);
    uint16_t res = vd == vr;
...
    if (res) {
...
        new_pc += 4;
...
    }
    edge_triggered(avr, avr->pc, new_pc);
...
```

Listing 4.8: CPSE Instruction Emulation [51]

CPSE compares the values in two registers [52]. *get_vd5_vr5* is a macro that initializes the variables *vd* and *vr*. These variables store the values of the two registers that are compared. If both values are equal, the next instruction is skipped.

The *edge_triggered* call is the instrumentation that was added. *avr->pc* and *new_pc* are the source and destination address of the jump, respectively.

The fuzzer maintains a hash set of all observed jumps and calls since fuzzing started. Each element in this hash set is a <source address, destination address> pair/struct. The hash set data structure is from the third-party Collections-C library [103].

The *edge_triggered* function checks if the hash set contains the <*avr->pc*, *new_pc*> pair. If this is not the case, the current input increased the branch coverage. Thus, the current input is considered a previous interesting input. The <*avr->pc*, *new_pc*> pair/struct is added to the hash set.

In addition, a 'New Coverage' message is sent to the user UI server. The user UI server runs in a separate process. All messages to the user UI server are sent via sockets. The 'New Coverage' message includes the current input, as well as the source and destination addresses of the edge.

When the fuzzer resets, an Input struct for the current input is created and this Input is added to the previous interesting inputs array. An input is added to this array only once, even if the input triggers multiple edges for the first time since the start of fuzzing.

4.5 Identify interesting Program States via Data Values

The state implementation must remember one or multiple previous values of a variable in order to compare the current value with previous values to decide whether the value and therefore the input is interesting. The user specifies the memory location of the variable and when the comparison should take place (i.e. at what value of the program counter in the emulated SUT).

What value and whether one or multiple values must be remembered depends on when the value is interesting (i.e. one of new MAX, new MIN, or UNIQUE). For example, if the user specifies MAX, the implementation must only the store one value, namely the largest value observed so far at the specified memory location and time. On the other hand, if the user specifies UNIQUE, the implementation must store all previous values in a hash set.

The state implementation supports the use case where a user uses patch_instruction with add_state multiple times for different variables, or for the same value but at different times, or with a different MIN, MAX, or UNIQUE. For example, if the user uses the following user API calls:

```
patch_instruction(get_symbol_address("some_function", avr),
                  add_state,
                  create_state_patch("some_variable", UNIQUE, avr));
patch_instruction(get_symbol_address("some_function", avr),
                  add_state,
                  create_state_patch("some_other_variable", UNIQUE, avr));
```

Listing 4.9: add_state example

The state implementation must remember the set of previous *some_variable* values and the set of previous *some_other_variable* values. To do this, the implementation uses a dictionary called SUT_state. Each patch_instruction call with add_state adds one entry to this dictionary. A dictionary value is a set of previous values. The dictionary key for a value consists of when the check occurs, what variable is checked, and when the value is interesting.

The data type of the dictionary value depends on the key's 'when the value is interesting'. If this is set to:

- **UNIQUE**, the dictionary value is a hash set of all previous values. The dictionary key specifies where the value is stored and when the value is checked.

- **MIN** or **MAX**, the dictionary value is a 64-bit unsigned integer (uint64) that stores the smallest or largest value observed so far, respectively. Like UNIQUE, the dictionary key specifies where the value is stored and when the value is checked.

The *add_state* function arguments specify when the check occurs (when the *add_state* function is called, this is the current program counter of the SUT and this value is accessible via the avr struct), the address of the checked variable, and when the value is interesting. These arguments are a SUT_state dictionary key. When the *add_state* function is called, add_state retrieves a dictionary value via this key and the current value. For example, if 'when the value is interesting' is set to MAX and the current value is larger than the dictionary value, the dictionary value is set to the current value and the current input is marked as interesting. On the other hand, if 'when the value is interesting' is set to UNIQUE and the current value is not in the dictionary value hash set, the hash set is updated with the current value and the current input is marked as interesting.

4.6 Crash Handler

The Crash Handler is notified when the emulator or a sanitizer detects a bug. Notify means that one of the functions in listing 4.10 is called:

```
1  void stack_buffer_overflow_found(avr_t *avr, avr_flashaddr_t crashing_addr) {
2    crash_found(avr, crashing_addr, 0, 0);
3  }
4
5  void uninitialized_value_used_found(avr_t *avr, avr_flashaddr_t crashing_addr,
6                                      avr_flashaddr_t origin_addr) {
7    crash_found(avr, crashing_addr, origin_addr, 1);
8  }
9
10 void timeout_found(avr_t *avr, avr_flashaddr_t crashing_addr) {
11   crash_found(avr, crashing_addr, 0, 2);
12 }
13
14 void invalid_write_address_found(avr_t *avr, avr_flashaddr_t crashing_addr) {
15   crash_found(avr, crashing_addr, 0, 3);
16 }
17
18 void bad_jump_found(avr_t *avr, avr_flashaddr_t crashing_addr) {
19   crash_found(avr, crashing_addr, 0, 4);
20 }
21
22 void reading_past_end_of_flash_found(avr_t *avr,
23                                      avr_flashaddr_t crashing_addr) {
24   crash_found(avr, crashing_addr, 0, 5);
25 }
26
27 void crash_found(avr_t *avr, avr_flashaddr_t crashing_addr,
28                  avr_flashaddr_t origin_addr, uint8_t crash_id) {
```

```
29 ...
```

Listing 4.10: Crash Handler Notification Functions

The main difference is that depending on the type of bug found, the *crash_found* function is called with different *crash_id* parameter values. In addition, if the bug is due to a UUM, the origin address is passed to *crash_found* as well. The *crashing_addr* is the program counter at the time when the sanitizer or emulator detects the bug/crash.

The implementation for crash deduplication is similar to detecting increases in code coverage. Crash Handler maintains a hash table called *unique_crashes*, whose keys are structs called *crash*. *crash* structs have three members: <*crashing_addr*, *origin_addr*, *crash_id*>. When the *crash_found* function is called, a new *crash* struct is initialized with the *crash_found* function parameter values. If the *unique_crashes* hash table does not contain this new *crash* struct, the combination of <*crashing_addr*, *origin_addr*, *crash_id*> values is found for the first time, the notification is classified as unique, and a 'New Crash' message is sent to the user UI server via sockets. In addition, new *crash* structs are added to the *unique_crashes* hash table. The hash table data structure is from the third-party Collections-C library [103].

A 'New Crash' message includes the stack trace at the time when the crash is detected. The program counters of the stack frames are stored in the stack_frame array, which was described in section 4.2.1. The stack trace information in the 'New Crash' message consists of the elements, i.e. the program counters, in the *stack_frame* [102]. Later, the user UI server converts the program counters to human readable source code line numbers and file names.

4.7 Emulator Reset

Simavr already implements resetting the emulator as part of its watchdog timer in 'System Reset' mode implementation. A system reset occurs when the *avr_reset* is called [106]. This resets the program counter, stack pointer, interrupts, timers, and more.

The *avr_reset* was extended to additionally reset any relevant fuzzer and sanitizer state. For example, the shadow map for the stack from the UUM sanitizer is reset as well.

4.8 User UI

The user UI server listens for incoming messages from the emulator process. Each message consists of a fixed-size header and a variable-length body. The header contains a field for the message ID and another field for the length of the message body. A message ID specifies the type of message. For example, a message with ID 1 is a 'New Coverage' message and ID 2 is a 'New Crash' message.

Messages with different IDs have different message body lengths. For example, the message body of 'New Crash' and 'New Coverage' messages contains the input that crashed or increased coverage. Inputs have different lengths.

The server must know when all bytes of a message are received and what bytes are part of the same message. The server knows when all bytes of a message are received via the body length field in the message header. As soon as all bytes of a message are received, the Parser component deserializes the message bytes to python objects and forwards them to a function from the Process Messages component. Which function from Process Messages is called depends on the type of message.

4.8.1 Crash Message

One function from the Process Messages component is *crashing_input*. The parser calls this function if the message ID is 'New Crash'. Deserialized python objects from the message are passed via function arguments. The source code for this function is shown in listing 4.11.

```
def crashing_input(self, crash_ID, crash_addr, origin_addr, crash_input,
    stack_frames_pc):
  # Found a new crash, write the stacktrace and the crashing input to a file
  stack_frame_text = 'Stackframe at crash:\n'
  for i, pc in enumerate(stack_frames_pc):
    stack_frame_text += f'    #{i} 0x{pc:x} in '
    stack_frame_text += self.addr_to_src(
        self.path_to_emulated_executable, pc, function_name=True
    ).decode('UTF-8').strip()
    stack_frame_text += '\n'

  self.save_crashing_input(
      crash_ID, crash_addr, origin_addr, crash_input, stack_frame_text)
```

Listing 4.11: crashing_input Function

The *crashing_input* function sets up *stack_frame_text*, which is later written to a file in the *save_crashing_input* function. The *stack_frames_pc* function parameter is a list of stack frame addresses. *stack_frame_text* includes these addresses in hex format and it includes the source code locations, i.e. the paths to the source code files and line numbers, of these addresses.

The conversion from address to source code location is implemented in the *addr_to_src* function, whose implementation is shown in listing 4.12.

```
def addr_to_src(self, path_to_emulated_executable: str, addr: int, function_name
    =False) -> str:
  cmd = ['avr-addr2line', '-e', path_to_emulated_executable, hex(addr)]
  if function_name:
    cmd += ['--functions', '--pretty-print']
  out = subprocess.check_output(cmd, timeout=10)
  return out
```

Listing 4.12: addr_to_src Function

The *addr_to_src* function implementation uses the third-party *avr-addr2line* tool [92]. *avr-addr2line* requires the debugging information in the executable file that the emulator is emulating. The user specifies the path to this file via command line arguments when the user starts the emulator. When the emulator starts, this file path is sent from the emulator process to the user UI server process via the 'Initial SUT information' message. The Process Messages component stores this path in the *path_to_emulated_executable* variable and this variable is passed to the *addr_to_src* function and the *avr-addr2line* tool.

In addition to storing the *stack_frame_text* in a file, the *save_crashing_input* function (called in listing 4.11) also stores the input that crashed the SUT in a separate file.

4.8.2 Coverage Message

The most interesting part of the Process Messages implementation for 'New Coverage' messages is the integration of the third-party gcovr tool [95]. Gcovr is typically used in conjunction with the output files from gcc's coverage instrumentation. The typical gcovr workflow is as follows:

1. The user compiles the SUT with gcc and the gcc command line flag '–coverage'. With this flag, gcc adds coverage instrumentation to the SUT binary. Similarly to the code coverage instrumentation that was implemented in this project, gcc adds code to all program branches that tracks how often the branch is executed.
2. When a process that is instrumented with gcc's coverage tracking code is executed and exits, the information about what source code lines were executed is stored in files. One such file is created for each source code file.
3. Gcovr reads the code coverage information from these files and creates for example an HTML representation of the coverage information.

This typical workflow only works when the SUT runs in an OS like Linux, because the instrumentation uses system calls to create files and to write the reached coverage information into these files. When compiling a SUT for Arduino, step 1 of the typical gcovr workflow above fails due to 'undefined reference' errors for the instrumentation.

For this reason, step 1 and 2 of the typical gcovr workflow must be adapted for the simavr gcovr workflow. Similar to step 1, i.e. gcc coverage instrumentation, the emulator collects coverage for the coverage-guided fuzzer. This coverage information is also sent to the user UI server.

Each 'New Coverage' message contains the information about one triggered edge. The sum of all 'New Coverage' messages received until fuzzing started is similar to the information that gets stored into files in step 2 of the typical gcovr workflow. For this reason, step 2 of the typical gcovr workflow can be replaced by writing the coverage information from 'New Coverage' messages to files in the format expected by gcovr. This replacement is implemented by the Process Messages component for 'New Coverage' messages.

An example output file from step 2 of the typical gcovr workflow is show in the following listing 4.13.

```
    -:    0:Source:path/to/src.c
    -:    0:Graph:src.gcno
    -:    0:Data:src.gcda
    -:    0:Runs:1
    -:    0:Programs:1
    -:    1:int main (void)
    1:    2:{
    1:    3:  return 0;
    -:    4:}
```

Listing 4.13: GCC Coverage Instrumentation Output File

In listing 4.13, the first 'column' specifies how often a program location was executed. If an entry in the second column starts with a *0*, the entry contains metadata, such as the path to the source code file. Otherwise, the entry specifies a source code line.

Gcovr does not use all the information in listing 4.13 to create an HTML coverage representation. Unused information can be omitted in the file. Listing 4.14 omits all data from listing 4.13 that is not used by gcovr.

```
    -:    0:Source:path/to/src.c
    1:    2:
    1:    3:
```

Listing 4.14: Required Data for Gcovr

Gcovr requires the path to the source code file and the source code line numbers of executed lines. In other words, gcovr requires source code locations of executed lines.

Each 'New Coverage' message includes a destination address of a jump. The instructions that follow this jump are linearly followed until a branch or return instruction is encountered. These following instructions must be marked as 'executed' as well. Instructions are disassembled via the third-party disassembly library Amoco [96].

The addresses of the jump and the instructions that follow until a branch or return instruction is encountered are converted from instruction address to source code location. This is implemented via the *addr_to_src* function (listing 4.12), which uses the third-party *avr-addr2line* tool [92].

When a 'New Coverage' message is received, files for gcovr are created and/or updated. Gcovr is run periodically.

5 Student Exercise

The exercises will introduce participants to fuzz test software that runs in an emulator. No prior experience with fuzzing and/or with emulators is required. Beginner-level experience with C/C++ and the Unix shell is recommended. To introduce what fuzzing and mutation-based fuzzing is, it is recommended that the participants read section 2.1 of this paper.

This chapter describes how to set up the virtual machine (VM). The text for the exercises is in a file in this VM.

Download and import the VM in VirtualBox. It is recommended that the VM has two or more processor cores available. When importing the VM with unchanged import settings, the VM should have 2 cores configured already. If the VM is imported already, you can check and set the number of cores in VirtualBox Manager via: Select the VM -> Machine -> Settings -> System -> Processor -> Processor(s) slide.

Start the VM. The credentials for this VM are:

- Username: user
- Password: 123456

In the VM, open the local file '/home/user/Exercises/API_docs/index.html' in a web browser. This displays the exercises, together with hints, additional information, and possible solutions to the exercises.

6 Evaluation and Conclusion

This work presented the design and implementation of an emulator fuzzer framework that integrated a mutation-based coverage-guided graybox fuzzer into the simavr emulator. In addition, exercises were created that introduce the participants to fuzz test microcontroller software that runs in simavr via the implemented emulator fuzzer framework.

Microcontroller software can read input from different sources. In fuzzing, this input must come from the fuzzer instead. The framework provides a C API where a user can specify how the input from the fuzzer is injected into the emulation. This approach is flexible, and it supports several use cases, for example if the SUT expects input via DMA, DMA followed by an external interrupt, or via function arguments or function return values such as the *Serial.read()* Arduino standard library function.

To detect errors (i.e. bugs), simavr had already implemented sanitizers to detect and report invalid opcodes, invalid read and write accesses, and the execution of non-existent instructions [51]. Additional sanitizers have been implemented and added to simavr's error detection capabilities to detect more and different types of errors. Specifically, sanitizers have been implemented and integrated into simavr that can detect stack buffer overflows, the use of uninitialized memory, and timeouts.

When a sanitizer detects a non-recoverable error, the emulator resets and restarts the SUT program with a new input from the fuzzer. Additionally, the user can configure when to reset via the C API.

The implemented reset mechanism is based on the implementation of the 'System Reset' mode from the watchdog timer. This watchdog timer implementation existed already and was not implemented as part of this work. The watchdog timer implementation was extended to also reset relevant state in the fuzzer and sanitizers. For example, the extended watchdog timer implementation can also reset the shadow map from the UUM sanitizer.

Instrumentation to all conditional jump instruction was added to collect branch coverage of executed inputs. This information is used by the coverage-guided fuzzer to prioritize previous inputs that increased the branch coverage. This fuzzer was also implemented as part of this work. The fuzzer uses the mutator from libFuzzer.

The coverage-guided approach fails for example when the SUT uses complex state machines. State machines are a common design pattern in embedded systems [86]. To aid the fuzzer in these cases, the C API allows the user to specify the location of data that represents the internal state of the SUT. The user also specifies when a internal state value is interesting. The fuzzer is notified if an interesting internal state is observed.

A User UI provides the user with information about the status of a fuzzing run. This information includes for example coverage information via the Coverage Explorer and graphs, as well as inputs and stack traces that triggered an error in the SUT.

List of Figures

Abbreviations

ASAN AddressSanitizer

DMA Direct Memory Access

IO Input/Output
IoT Internet of Things
IPC Inter Process Communication
IR Intermediate Representation

MMU Memory Management Unit

QASan QEMU-AddressSanitizer

SUT Software under Test

TSAN Thread Sanitizer

UBSAN Undefined Behavior Sanitizer
UUM Uninitialized Memory

VM Virtual Machine

Bibliography

[1] 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)*. USENIX Association, Aug. 2020, pp. 1237–1254. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/feng.

[2] Marcel Boehme, Cristian Cadar, and Abhik ROYCHOUDHURY. "Fuzzing: Challenges and Reflections". In: *IEEE Software* 38.3 (2021), pp. 79–86. DOI: 10.1109/MS.2020.3016773.

[3] Valentin J. M. Manes et al. *The Art, Science, and Engineering of Fuzzing: A Survey*. 2019. arXiv: 1812.00140 [cs.CR].

[4] 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)*. Washington, D.C.: USENIX Association, Aug. 2015. URL: https://www.usenix.org/conference/woot15/workshop-program/presentation/koscher.

[5] Michel Pollet et al. *Github simavr*. 2021. URL: https://github.com/buserror/simavr (visited on 05/02/2021).

[6] Yaowen Zheng et al. "FIRM-AFL: High-Throughput Greybox Fuzzing of Iot Firmware via Augmented Process Emulation". In: *Proceedings of the 28th USENIX Conference on Security Symposium*. SEC'19. Santa Clara, CA, USA: USENIX Association, 2019, pp. 1099–1114. ISBN: 9781939133069.

[7] Aljoscha Lautenbach. *On Cyber-Security for In-Vehicle Software*. 2021. URL: https://publications.lib.chalmers.se/records/fulltext/500039/500039.pdf (visited on 06/10/2021).

[8] Jack Erjavec. *Automotive technology : a systems approach*. Clifton Park, NY, USA: Cengage Learning, 2015. ISBN: 978-1133612315.

[9] NIST. *NIST CVE Statistics Results - Total Matches By Year*. 2021. URL: https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&search_type=all (visited on 06/10/2021).

[10] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution". In: Jan. 2016. DOI: 10.14722/ndss.2016.23368.

[11] Aparna Dey. *Combining Manual with Automation Testing*. 2020. URL: https://blogs.perficient.com/2020/07/31/combining-manual-with-automation-testing/ (visited on 06/11/2021).

[12] Kyriakos Ispoglou et al. “FuzzGen: Automatic Fuzzer Generation”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2271–2287. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou.

[13] Marcel Böhme. *Marcel Böhme Twitter*. 2021. URL: https://twitter.com/mboehme_/status/1362901814450941952 (visited on 06/10/2021).

[14] Patrice Godefroid. *SAGE: Whitebox Fuzzing for Security Testing*. 2012. URL: https://queue.acm.org/detail.cfm?id=2094081 (visited on 06/11/2021).

[15] Brandon Falk. *Twitter AFL++ Benchmark*. 2020. URL: https://twitter.com/gamozolabs/status/1265481886769414144?lang=en (visited on 05/12/2021).

[16] Barton P. Miller, Louis Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[17] Michał Zalewski. *AFL Github*. 2021. URL: https://github.com/google/AFL (visited on 04/18/2021).

[18] Cunningham et al. *Sidewinder - An Evolutionary Guidance System for Malicious Input Crafting*. 2006. URL: https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Embleton.pdf (visited on 06/11/2021).

[19] Andrea Fioraldi et al. “AFL++ : Combining Incremental Steps of Fuzzing Research”. In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[20] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. “Coverage-Based Greybox Fuzzing as Markov Chain”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1032–1043. ISBN: 9781450341394. DOI: 10.1145/2976749.2978428. URL: https://doi.org/10.1145/2976749.2978428.

[21] Marc Heuse et al. *AFLplusplus - Custom Mutators*. 2021. URL: https://github.com/AFLplusplus/AFLplusplus/tree/stable/custom_mutators (visited on 05/08/2021).

[22] Chenyang Lyu et al. “MOPT: Optimized Mutation Scheduling for Fuzzers”. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu.

[23] Jiongyi Chen et al. “IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing”. In: *NDSS*. 2018.

[24] Xiaotao Feng et al. *Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference*. 2021. arXiv: 2105.05445 [cs.CR].

[25] Joshua Pereyda. *boofuzz: Network Protocol Fuzzing for Humans*. 2021. URL: https://boofuzz.readthedocs.io/en/stable/ (visited on 06/12/2021).

[26] Microchip. *ATmega2560*. 2021. URL: https://www.microchip.com/wwwproducts/en/ATmega2560 (visited on 05/16/2021).

[27] Andrea Biondo. *Coverage-guided fuzzing of embedded firmware with avatar2*. 2021. URL: https://siagas.math.unipd.it/siagas/getTesi.php?id=2030 (visited on 06/12/2021).

[28] Marius Muench et al. "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices". In: *NDSS*. 2018.

[29] Fabrice Bellard. *QEMU*. 2021. URL: https://www.qemu.org/ (visited on 06/13/2021).

[30] Yaowen Zheng et al. "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1099–1114. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/zheng.

[31] Tim Newsham et al. *TriforceAFL*. 2017. URL: https://github.com/nccgroup/TriforceAFL (visited on 06/13/2021).

[32] Nathan Voss. *afl-unicorn*. 2020. URL: https://github.com/Battelle/afl-unicorn (visited on 06/14/2021).

[33] Nguyen Anh Quynh et al. *Unicorn Engine*. 2021. URL: https://github.com/unicorn-engine/unicorn (visited on 06/14/2021).

[34] Nathan Voss. *afl-unicorn: Fuzzing Arbitrary Binary Code*. 2017. URL: https://medium.com/hackernoon/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf (visited on 06/14/2021).

[35] Dominik Maier, Benedikt Radtke, and Bastian Harren. "Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing". In: *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA: USENIX Association, Aug. 2019. URL: https://www.usenix.org/conference/woot19/presentation/maier.

[36] Marc Heuse et al. *AFLplusplus README*. 2021. URL: https://github.com/AFLplusplus/AFLplusplus (visited on 06/15/2021).

[37] Marc Heuse et al. *AFLplusplus Fuzzing with Unicorn Mode*. 2021. URL: https://github.com/AFLplusplus/AFLplusplus/blob/stable/unicorn_mode/README.md#fuzzing-with-unicorn-mode (visited on 06/15/2021).

[38] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. "Fuzzing Binaries for Memory Safety Errors with QASan". In: *2020 IEEE Secure Development (SecDev)*. 2020, pp. 23–30. DOI: 10.1109/SecDev45635.2020.00019.

[39] Brandon Falk. *fuzz_with_emus Github*. 2021. URL: https://github.com/gamozolabs/fuzz_with_emus (visited on 06/16/2021).

[40] Pengfei Wang et al. *The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing*. 2021. arXiv: 2005.11907 [cs.CR].

[41] Patrice Godefroid. "Fuzzing: Hack, Art, and Science". In: *Commun. ACM* 63.2 (Jan. 2020), pp. 70–76. ISSN: 0001-0782. DOI: 10.1145/3363824. URL: https://doi.org/10.1145/3363824.

[42] Andrea Fioraldi. *Program State Abstraction for Feedback-Driven Fuzz Testing using Likely Invariants*. 2021. URL: https://arxiv.org/pdf/2012.11182.pdf (visited on 06/17/2021).

[43] Google, Inc. *libFuzzer – a library for coverage-guided fuzz testing*. 2021. URL: https://llvm.org/docs/LibFuzzer.html (visited on 04/18/2021).

[44] Google, Inc. *AddressSanitizer*. 2019. URL: https://github.com/google/sanitizers/wiki/AddressSanitizer (visited on 04/29/2021).

[45] Marcel Böhme et al. *Mutation-Based Fuzzing*. 2021. URL: https://www.fuzzingbook.org/html/MutationFuzzer.html# (visited on 06/18/2021).

[46] Gary J Saavedra et al. *A Review of Machine Learning Applications in Fuzzing*. 2019. arXiv: 1906.11133 [cs.CR].

[47] Marcel Böhme et al. *Greybox Fuzzing*. 2021. URL: https://www.fuzzingbook.org/html/GreyboxFuzzer.html (visited on 06/18/2021).

[48] Alexander Kössler. *qsimavr - Graphical Simulation of an AVR Processor and Periphery*. 2012. URL: https://github.com/schuay/bachelors_thesis/blob/master/thesis.pdf (visited on 07/01/2021).

[49] Victor Moya del Barrio. *Study of the techniques for emulation programming*. 2001. URL: http://www.xsim.com/papers/Bario.2001.emubook.pdf (visited on 07/01/2021).

[50] E. Roberts. *RISC Architecture - How Pipelining Works*. 2021. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/pipelining/index.html (visited on 07/01/2021).

[51] Michel Pollet et al. *Github simavr source code - sim_core*. 2021. URL: https://github.com/buserror/simavr/blob/f1aad44bce3aba57ba01083a46b79f4731e9dda9/simavr/sim/sim_core.c (visited on 05/02/2021).

[52] Microchip. *AVR Instruction Set Manual*. 2016. URL: http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf (visited on 06/21/2021).

[53] Michel Pollet et al. *Github simavr source code - sim_avr.c avr_callback_run_raw function*. 2021. URL: https://github.com/buserror/simavr/blob/1d227277b3d0039f9faef9ea62880ca3051b14f8/simavr/sim/sim_avr.c#L359 (visited on 07/01/2021).

[54] Michel Pollet et al. *Github simavr source code - sim_avr.c*. 2021. URL: https://github.com/buserror/simavr/blob/1d227277b3d0039f9faef9ea62880ca3051b14f8/simavr/sim/sim_avr.c (visited on 07/01/2021).

[55] Michał Zalewski. *AFL Github - README fuzzing binaries*. 2021. URL: https://github.com/google/AFL#6-fuzzing-binaries (visited on 04/21/2021).

[56] Joshua Pereyda. *boofuzz Github*. 2021. URL: https://github.com/jtpereyda/boofuzz/tree/a23b3ddb4cac70476d6b1cbcb3eacf5fe673b8b9 (visited on 05/15/2021).

[57] GJLay et al. *avr-gcc*. 2021. URL: https://gcc.gnu.org/wiki/avr-gcc (visited on 07/02/2021).

[58] Thomas Gruber and Daniel Schuroff. *Compiler Sanitizers*. 2019. URL: https://hpc-wiki.info/hpc/Compiler_Sanitizers (visited on 04/22/2021).

[59] Google, Inc. *sanitizer*. 2021. URL: https://github.com/google/sanitizers (visited on 04/28/2021).

[60] Google, Inc. *UndefinedBehaviorSanitizer*. 2021. URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html (visited on 05/01/2021).

[61] Alexander Entinger et al. *Build Clang/LLVM for AVR*. 2020. URL: https://lists.llvm.org/pipermail/llvm-dev/2020-March/140323.html (visited on 06/16/2021).

[62] LLVM Developers. *LLVM AVR backend*. 2021. URL: https://llvm.org/doxygen/md_lib_Target_AVR_README.html (visited on 06/16/2021).

[63] Paul M. Bendixen et al. *avr-libstdcxx*. 2019. URL: https://gitlab.com/avr-libstdcxx/gcc (visited on 05/01/2021).

[64] Zhi Zhang et al. *Standard C++ library on GCC's AVR target*. 2019. URL: https://www.reddit.com/r/cpp/comments/esk2wy/standard_c_library_on_gccs_avr_target/ (visited on 05/01/2021).

[65] Julian Seward. *Valgrind Supported Platforms*. 2021. URL: https://www.valgrind.org/info/platforms.html (visited on 05/02/2021).

[66] Alok Save. *GCC -Wuninitialized / -Wmaybe-uninitialized issues*. 2013. URL: https://stackoverflow.com/questions/14132898/gcc-wuninitialized-wmaybe-uninitialized-issues/14132910#14132910 (visited on 05/02/2021).

[67] ISO. *ISO/IEC 14882:2013: Programming languages — C++*. 2013.

[68] Evgeniy Stepanov and Konstantin Serebryany. "MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++". In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '15. San Francisco, California: IEEE Computer Society, 2015, pp. 46–55. ISBN: 9781479981618.

[69] LLVM Project. *LLVM Language Reference Manual - Introducton*. 2021. URL: https://llvm.org/docs/LangRef.html#introduction (visited on 05/01/2021).

[70] Brandon Falk. *Github fuzz_with_emus emulator.rs*. 2021. URL: https://github.com/gamozolabs/fuzz_with_emus/blob/22e791c12b2588ee01e7517a650b7ef49928b7bc/src/emulator.rs#L362 (visited on 05/02/2021).

[71] Arduino Developers. *Arduino Libraries*. 2021. URL: https://www.arduino.cc/en/reference/libraries (visited on 05/02/2021).

[72] Google, Inc. *ThreadSanitizer*. 2021. URL: https://clang.llvm.org/docs/ThreadSanitizer.html (visited on 05/03/2021).

[73] Michel Pollet et al. *Github simavr source code - Stack Buffer Overflow Implementation*. 2021. URL: https://github.com/buserror/simavr/blob/f1aad44bce3aba57ba01083a46b79f4731e9dda9/simavr/sim/sim_core.c#L142 (visited on 05/02/2021).

[74] Konstantin Serebryany et al. “AddressSanitizer: A Fast Address Sanity Checker”. In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, June 2012, pp. 309–318. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[75] Michel Pollet et al. *Github simavr source code - sim_avr.h address size*. 2021. URL: https://github.com/buserror/simavr/blob/1d227277b3d0039f9faef9ea62880ca3051b14f8/simavr/sim/sim_avr.h#L172 (visited on 05/02/2021).

[76] Richard Stallman et al. *avr-gcc(1) - Linux man page*. 2021. URL: https://linux.die.net/man/1/avr-gcc (visited on 05/05/2021).

[77] Google, Inc. *MemorySanitizer*. 2021. URL: https://github.com/google/sanitizers/wiki/MemorySanitizer (visited on 05/06/2021).

[78] Julian Seward. *Memcheck: a memory error detector*. 2021. URL: https://valgrind.org/docs/manual/mc-manual.html (visited on 05/06/2021).

[79] Platonov Sergey. *Address/Thread/Memory Sanitizer*. 2015. URL: https://www.slideshare.net/sermp/sanitizer-cppcon-russia (visited on 05/06/2021).

[80] Mathias Payer. *CS412 Software Security - Program Testing - Fuzzing*. 2019. URL: https://nebelwelt.net/teaching/19-412-SoSe/slides/11-fuzzing.pdf (visited on 05/08/2021).

[81] Yanhao Wang et al. “Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization”. In: *NDSS*. 2020.

[82] Shuitao Gan et al. “CollAFL: Path Sensitive Fuzzing”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 679–696. DOI: 10.1109/SP.2018.00040.

[83] *Branch Coverage*. URL: https://www.cs.odu.edu/~cs252/Book/branchcov.html (visited on 05/09/2021).

[84] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World’s Most Popular Disassembler*. USA: No Starch Press, 2011. ISBN: 1593272898.

[85] Cornelius Aschermann et al. “Ijon: Exploring Deep State Spaces via Fuzzing”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1597–1612. DOI: 10.1109/SP40000.2020.00117.

[86] Deepti Mani. *Implementing finite state machines in embedded systems*. 2016. URL: https://www.embedded.com/implementing-finite-state-machines-in-embedded-systems/ (visited on 07/01/2021).

[87] George Klees et al. “Evaluating Fuzz Testing”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 2123–2138. ISBN: 9781450356930. DOI: 10.1145/3243734.3243804. URL: https://doi.org/10.1145/3243734.3243804.

[88] Oeyvind A. Sandberg. *AVR Watchdog Timer*. 2021. URL: https://microchipdeveloper.com/8avr:avrwdt (visited on 05/12/2021).

[89] Microchip. *Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V Datasheet*. 2014. URL: https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf (visited on 05/12/2021).

[90] Marc Heuse et al. *AFLplusplus - Deferred initialization*. 2021. URL: https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.persistent_mode.md#3-deferred-initialization (visited on 05/12/2021).

[91] Max Reitz. *QEMU Backups (and snapshots) with QEMU*. 2016. URL: https://www.linux-kvm.org/images/6/65/02x08B-Max_Reitz-Backups_with_QEMU.pdf (visited on 05/12/2021).

[92] Michael Kerrisk. *addr2line(1) — Linux manual page*. 2021. URL: https://man7.org/linux/man-pages/man1/addr2line.1.html (visited on 05/06/2021).

[93] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[94] Michael L. Waskom. "seaborn: statistical data visualization". In: *Journal of Open Source Software* 6.60 (2021), p. 3021. DOI: 10.21105/joss.03021. URL: https://doi.org/10.21105/joss.03021.

[95] William Hart et al. *gcovr Github*. 2021. URL: https://github.com/gcovr/gcovr (visited on 05/14/2021).

[96] Axel Tillequin. *amoco*. 2021. URL: https://github.com/bdcht/amoco (visited on 06/25/2021).

[97] Troy D. Hanson. *uthash*. 2021. URL: https://github.com/troydhanson/uthash (visited on 06/25/2021).

[98] Michel Pollet et al. *Github simavr source code - sim_elf.c*. 2021. URL: https://github.com/buserror/simavr/blob/master/simavr/sim/sim_elf.c (visited on 05/15/2021).

[99] Michel Pollet et al. *elfutils*. 2021. URL: https://sourceware.org/elfutils/ (visited on 05/15/2021).

[100] Jeremy Cole. *Architecture of the AVR*. 2010. URL: https://blog.jcole.us/2010/07/17/architecture-of-the-avr/ (visited on 05/15/2021).

[101] Scott Thornton. *What is the difference between Von-Neumann and Harvard architectures*. 2018. URL: https://www.microcontrollertips.com/difference-between-von-neumann-and-harvard-architectures/ (visited on 05/15/2021).

[102] Michel Pollet et al. *Github simavr source code - sim_avr.h stack_frame*. 2021. URL: https://github.com/buserror/simavr/blob/1d227277b3d0039f9faef9ea62880ca3051b14f8/simavr/sim/sim_avr.h#L133 (visited on 06/22/2021).

[103] Srdan Panic. *Github Collections-C*. 2021. URL: https://github.com/srdja/Collections-C (visited on 05/16/2021).

[104] Andrea Fioraldi et al. *AFL++ load_custom_mutator Function*. 2021. URL: https://github.com/AFLplusplus/AFLplusplus/blob/48cef3c74727407f82c44800d382737265fe65b4/src/afl-fuzz-mutators.c#L138 (visited on 07/01/2021).

[105] Michael Kerrisk. *dlopen(3) — Linux manual page*. 2021. URL: https://man7.org/linux/man-pages/man3/dlopen.3.html (visited on 06/22/2021).

[106] Michel Pollet et al. *Github simavr source code - sim_avr.h avr_reset function*. 2021. URL: https://github.com/gatk555/simavr/blob/master/simavr/sim/sim_avr.c#L196 (visited on 06/22/2021).