AVRS: Emulating AVR Microcontrollers for Reverse Engineering and Security Testing

Michael Pucher
SBA Research
Vienna, Austria

Christian Kudera
SBA Research
Vienna, Austria

Georg Merzdovnik
SBA Research
Vienna, Austria

ABSTRACT

Embedded systems and microcontrollers are becoming more and more popular as the Internet of Things continues to spread. However, while there is a wealth of different methods and tools for analyzing software and firmware for architectures that are common to standard hardware, such as x86 or Arm, other systems have not been scrutinized so closely. One of these widely used architectures are AVR 8-bit microcontrollers, which are also used in projects like the Arduino platform. This lack of tools makes it more difficult to analyze such systems and identify potential security vulnerabilities. To get the most out of modern reverse engineering and debugging techniques such as fuzzing or concolic execution, sophisticated and correct emulators are required for dynamic analysis.

The presented work tries to close this gap by introducing AVRS, a lean AVR emulator prototype developed with the goal of reverse engineering. It was implemented to overcome limitations in existing emulators, such as completeness or execution speed, and to provide simple interfaces for interaction with existing program analysis and reverse engineering tools. We provide an analysis of AVRS in relation to existing emulators and show the improvements in speed and completeness. In addition, we have created a setup that leverages AVRS to use fuzz tests to automatically identify errors in AVR firmware. Our results indicate that AVRS is a valuable addition to the arsenal of analysis tools for embedded firmware and can be easily extended to allow the use of existing analysis tools in the domain of AVR microcontrollers.

KEYWORDS

IoT, AVR, emulation, fuzzing, reverse engineering, security analysis, embedded systems

ACM Reference Format:
Michael Pucher, Christian Kudera, and Georg Merzdovnik. 2020. AVRS: Emulating AVR Microcontrollers for Reverse Engineering and Security Testing. In *The 15th International Conference on Availability, Reliability and Security (ARES 2020), August 25–28, 2020, Virtual Event, Ireland*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3407023.3407065

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ARES 2020, August 25–28, 2020, Virtual Event, Ireland
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8833-7/20/08...$15.00
https://doi.org/10.1145/3407023.3407065

1 INTRODUCTION

AVR 8-bit microcontrollers (MCUs) are used in automotive applications, sensor nodes and IoT devices, while also being popular with hobbyists. They are furthermore the basis for most boards of the Arduino product line, which in turn is used in a variety of projects, like data logging [48] or control and measurement systems [30], and is often used to bootstrap development of projects because of the platform's ease of use. This widespread usage reinforces the relevance of security: every time the MCU processes data received from peripherals, it is handling untrusted user input and due to relatively weak processing power, thorough input validation can easily become an afterthought. Most embedded firmware is written in C or C++, trading runtime boundary checks for performance and making it harder for a developer to write correct code. Therefore many of the encountered vulnerabilities in embedded devices can still be categorized as *easy targets*, like simple buffer overflows [1] or format string vulnerabilities.

On desktop systems, identification of such bugs usually happens with modern reverse engineering and software testing techniques like concolic execution or fuzz testing. However, those are only applicable if the underlying system under test can be efficiently monitored during execution, and analyzing MCU firmware images for vulnerabilities is different from desktop applications. Analysis of such systems is therefore sometimes conducted as a *trial-and-error* approach, to identify or verify potential vulnerabilities [3].

The need for specialized analysis methods of embedded and IoT firmware has already been shown by Muench et al. [28]. Their evaluation highlights the contrast to commodity systems, whereas embedded devices often only support a limited amount of monitoring capabilities to detect faulty states. While a desktop application will be terminated by the operating system if an invalid memory access occurs due to an overflow, an embedded device will continue running and might behave in unpredictable ways. This is especially problematic for automated fuzz testing, since it is hard for the fuzzer to identify if it actually triggered a bug or not. While there exists a range of disassemblers and static analysis tools implemented in notable reverse engineering tools (e.g. *Ghidra* [2] or *radare2* [5]) that can handle AVR 8-bit firmware, there are no easy-to-use tools for dynamic analysis and emulation readily available. In order to increase the overall security of AVR-powered devices, potent reverse engineering tools are required to find potential vulnerabilities in safety critical systems, as source code of those applications is usually not available to independent researchers and dynamic analysis techniques are still applicable, even if the source code is known.

Therefore our work presents AVRS, a new emulation environment for AVR 8-bit microcontrollers, which was specifically designed to enable reverse engineering and security testing of embedded systems. To identify restrictions of previous approaches, we provide an in-depth comparison of existing AVR emulators. This comparison covers topics like performance and completeness, concerning coverage of available devices and instruction sets. Our analysis shows that existing tools have certain drawbacks in these regards, which directly influenced the implementation of AVRS to overcome those limitations. To show the feasibility of combining AVRS with existing security testing tools we created a fuzz testing set up in combination with *boofuzz*, to identify security vulnerabilities, like stack overflows or format string vulnerabilities, in AVR firmware. To facilitate reproducibility and encourage openness, all created evaluation artifacts are published together with the source-code of AVRS and are available at the project website[1].

In particular, the main contributions of this paper are as follows:

Evaluation of Existing AVR Emulation Tools:
To evaluate the current state of AVR security analysis tools, we provide an overview of existing emulation tools and compare them concerning performance and completeness.

AVRS: Emulator focused on Security Analysis:
Based on these results, we present AVRS, an emulator designed to specifically overcome drawbacks in existing tools to allow for security testing of AVR based embedded devices.

Methodology for fuzz testing of AVR MCUs:
Using *boofuzz* together with AVRS, we implement a test setup to show the feasibility of our approach concerning security testing of AVR controllers.

2 BACKGROUND

Building tooling for dynamic analysis on AVR comes with the same issues as other microcontroller architectures, but the AVR architecture has certain specifics, which need to be taken care of when designing an emulator architecture.

Instruction Set Architecture. The ISA used by AVR microcontrollers is a RISC architecture, having 99 distinct opcodes after removing aliases, with the presence and semantics of instructions varying between different MCUs on certain characteristics. Different program counter sizes change the behavior of calls/returns, by storing the return address either with 2 or 3 bytes on the stack; behavior for returning from interrupts is different whether XMega or Mega devices have been used, and the Tiny MCUs use smaller equivalents for *LDS*/*STS* instructions, which overlap with instructions from other device families [13].

Memory Layout. The memory layout is dictated by the types of physical memory built into AVR MCUs, most notably flash memory for storing the executable code, SRAM for temporary data and usually EEPROM or persisting data across resets. Since SRAM is scarce, the program cannot be loaded into RAM for execution and is therefore executed from flash memory, resulting in a Harvard architecture. The flash memory consists of a contiguous chunk of memory for the program code, with a potentially available bootloader section. On the other hand, the data memory uses memory mappings to present different memories and interfaces to the MCU, with the exact layout depending on the device. In general, data memory will at least contain a section of memory mapped I/O ports, while more feature rich MCUs can map the registers, SRAM and device specific I/O into the data space. An example for this can be seen in Figure 1, showing the memory map of an XMega where EEPROM is mapped into data memory. Because accessing memory mapped I/O causes side effects other than manipulating memory, an emulator has to keep track of memory access in these regions.

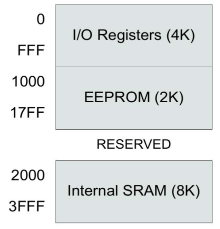

Figure 1: Memory layout of an ATXMega128a4u [12].

Interrupts. To respond to certain events, AVR firmware can assign interrupt service routines (ISR) by adding a jump to the routine in the interrupt vector table (IVT), and triggering an interrupt can then be modelled as a jump to the IVT entry. When emulating this, the semantics of an interrupt call need to be simulated: a pending interrupt is executed after the next instruction and interrupts are queued per interrupt, meaning an ISR will only be called once when the interrupt is triggered multiple times. Interrupt priority is being handled by the IV address according to the datasheet; interrupts with a lower IV address have priority over the ones with a higher address [24, p. 381].

Peripherals To communicate with the outside world, peripherals allow interaction by leveraging two channels: interrupts and memory mapped I/O registers. As every I/O register can be a source for malicious input, being able to attach custom analysis modules to, e.g. fuzz test a firmware implementation, should be one main goal of an emulator. The emulator has to watch for state changes in the I/O registers and be able to notify an external module of the state change. Conversely, an external module must be allowed to change I/O register states and trigger interrupts.

Firmware File Formats When dumping firmware from a device or through other means, it will likely be a BLOB without annotations. The preferred input format of an emulator is therefore raw binary or the Intel HEX format [20], which can be easily produced from raw binary. As existing emulators are designed for production and not reverse engineering, these rely on more descriptive formats, such as the Atmel ELF format [19].

3 RELATED WORK

Security analysis of Embedded Systems and IoT applications is not a new idea. In recent years, several studies already tried to shed light on the analysis of such applications, with surveys focusing on domains like commodity IoT applications [9] or home based IoT deployments [4]. The results show that the state of IoT security

[1]Project website: https://avrs.appsec.at

still offers room for improvement. This points out the importance for adequate methods and tools to test devices concerning security and privacy implications.

Large scale studies analysing the state of IoT device security mostly focus on specific samples (e.g. embedded Linux firmware distributions [10]), only parts of the firmware (e.g. web apps [14]) or focus on high level aspects of the system (e.g authentication bypass vulnerabilities [40]). What these studies have in common is their mostly limited set of supported architectures. While those methods work well for larger controllers, like ARM or MIPS running an embedded operating system, they do not support low powered architectures which might only run firmware directly. The following will provide better insights into the current state of security analysis for such embedded devices.

3.1 AVR Security

The security of AVR controllers has been scrutinized as well. A study of the Arduino Yun identified several vulnerabilities in the deployed firmware [3]. However, they had to resort to static analysis and a *trial-and-error* approach, because of the lack of available tools for security testing. While they were still able to identify vulnerabilities with this manual approach, this furthermore shows the need for automated testing environments, as manual analysis can be tedious and cannot be applied at scale. The topic of AVR emulation in particular first arose in an academic context within the field of sensor network simulation, trying to increase the robustness of such systems by allowing them to be simulated in the course of development. This started with TOSSIM [22], which simulated the TinyOS library calls made by the firmware, followed by atemu [33] and Avrora [43], which began to emulate the AVR core itself, to provide a more accurate and versatile simulation result. Further accuracy improvements, such as cycle-accuracy of cryptographic primitives or emulation under real-time constraints have been implemented in other simulation frameworks [21, 37, 49]. None of these projects have been designed with dynamic analysis of unknown firmware BLOBs in mind, and there are hardly any features allowing for introspection of emulator state or extensibility, with the exception of Avrora, which allows for custom probes and memory watches being executed as a plugin during simulation runtime [45].

3.2 Analyzing Embedded Devices

There already exist a variety of methods for dynamic testing of IoT devices, like cross program taint analysis for IoT systems [23] to multi target orchestration platforms like Avatar^2 [26].

In this paper we focus on fuzz testing as an example use case for AVRS. Fuzz testing has already been employed for a variety of use-cases and has been a heavily researched topic in recent years, enhancing existing methodologies but also bringing forth new areas of fuzzing, like algorithmic denial-of-service vulnerabilities [7]. IoTFuzzer [11] extracts protocol information for IoT devices from smartphone companion apps to fuzz IoT devices. Therefore it cannot be used in contexts where no such application is available. Muench et al. [28] already provide results where they showed that emulation based approaches yield higher throughput compared to directly fuzzing IoT devices, as commodity hardware usually runs magnitudes faster than the real device. FIRM-AFL [50] by Zheng et al. extends on this assumption and provides a system for high-throughput fuzzing of embedded systems by combining system mode emulation and user mode emulation to enhance performance. FIRMCORN [16] proposes to improve fuzzing of embedded devices by optimizing virtual execution of devices. As these recent results show, emulation of embedded firmware is a viable approach to the security analysis of embedded devices, raising the importance of available emulators for a diverse set of architecture.

4 METHODOLOGY

Due to limitations in existing emulators, a new emulator, called AVRS, is being implemented. The aim of the implementation phase is not to add as many new AVR cores and peripherals as possible, but to create a lean base system that is extensible and can be used on a given firmware binary without extensive manual effort. This phase is split into three parts: constructing an instruction decoder and emulator, the emulation manager taking care of instrumenting the instruction emulator and linking it with peripherals, as well as a graphical user interface. As the goal of this implementation is to improve upon existing emulators, an evaluation of supported processor features, correctness and performance is performed in comparison to a selection of emulators. To be included in this selection an emulator has to fulfill several criteria, which ensure that an emulator implementation is sufficient for further research to build upon, which is one goal of AVRS. The emulator source code has to be available, as without this, it cannot be extended in case certain features are missing. An emulator has to implement more than one AVR core, or a mechanism to add more different cores; this way, only emulators with a certain level of maturity are being considered and emulators which have been designed for one special purpose are being avoided. Finally, the emulator should provide rudimentary documentation, or examples, and has to be buildable on a modern system. Based on these criteria, the following emulators have been selected:

(1) Avrora [43, 44]
(2) simavr [32]
(3) SimulAVR [36]
(4) atemu [33, 34]
(5) GNU AVR Simulator [46]
(6) IMAVR [17]

Out of these, only Avrora published performance benchmarks, which are not reproducible, as the code used for the benchmarks has not been published. However, we still try to estimate a comparison with these numbers, by including the same Livermore loop benchmarks [31] in this evaluation. Additionally, computation intensive benchmarks are performed by running cryptographic algorithms from pre-existing libraries [8, 18].

We chose to implement AVRS from scratch in order to avoid fixation on a specific core family, as the listed emulators have been mostly designed to work with devices of the Mega family. As such, large updates in the memory access parts are required and writing it from scratch allows a simpler design. Since we also want AVRS to serve as the base for a tightly integrated reverse engineering environment, rewriting allows us to explore different ideas, such as using an efficient intermediate representation for execution/disassembly. Applying this to any of the listed emulators would require a rewrite of their core emulation primitives, but without the benefit of having a minimal codebase.

Additionally, a patch set [35] introduced the AVR architecture to QEMU [42]. We chose not to include this implementation in the selected emulators, as it is still experimental and being reworked to be merged upstream. While this might be a promising solution, QEMU does not offer cycle-accurate emulation, whereas AVRS and the other listed emulators provide this. We also believe that an emulator implementing AVR, as a relatively simple architecture, does not benefit from the complexity of the QEMU codebase.

5 AVRS

AVRS has been designed to reuse existing approaches for common tasks found in existing emulators, while trying to improve upon features that have been found lacking. While most other emulators have been implemented in C, AVRS has been written in Rust, allowing low-level memory-safe programming, outperforming garbage collected languages. The frontend makes use of C++ and Qt to ensure a stable cross-platform GUI implementation, as Rust GUIs are limited.

5.1 Overview

The architecture of AVRS is split in four separate parts, outlined in Figure 2: The core, the emulation manager, the GUI and a set of utilities. This architecture ensures reusability of small components and a clean interface for the GUI code to communicate with. The core part is responsible for decoding instructions into an intermediate representation (IR), a superset of instruction variants across all AVR cores, and executing the IR instructions. No memory is allocated in this part, leaving the memory management to the caller.

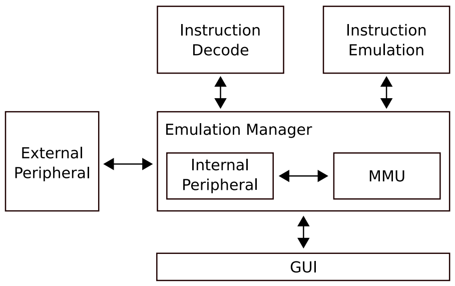

Figure 2: Overview of the macro architecture of AVRS.

The core stores a minimal amount of state: the program counter, stack pointer, status register, emulation state and interrupt management information. All other information is stored in a contiguous block of memory and accessed with the offsets of the MCU core memory map being used. Memory sections other than the register file and SRAM may perform operations on read/write, which is relevant for memory mapped I/O. Tracking memory access for I/O mapped memory is required for peripheral support and with the core kept at a minimum, this information has to be managed at the emulation manager layer. To this end, every module registered in the emulation manager can read and modify the emulator state after one instruction step, while also being able to hook memory access functions used by the core.

The emulation manager is a library that builds on the core and can act as a standalone emulator. The manager is taking care of running the emulation loop at a specified speed and relaying effects of memory manipulation to other components. One of these components is an interface for allowing external control of the emulation system, similar to the way the emulation manager controls the core; this provides an interface to implement external programs, and is being leveraged by the GUI component of AVRS.

Another aspect of the emulation manager is handling of peripherals, i.e. components on or connected to the MCU, which are being controlled via memory mapped I/O and interrupts. In an attempt to provide a clear-cut interface for all different types of peripherals, AVRS provides an interface to implement peripherals in a modular way, taking the following issues into account:

(1) **Interrupts**: Peripherals need the ability to trigger interrupts in response to interaction with the emulator or events which occured during state changes.
(2) **I/O Register Read/Write**: Communication with peripherals is achieved by changing values in memory mapped I/O registers, and they can provide data when reading from these registers. In the case of I/O registers, both, reads and writes can cause state changes to the internal peripheral logic. An example for this is a UART controller, as found on most AVR MCUs: Because received data is stored in a queue, reading the value from the data register using a load instruction causes the next byte in the queue to shift into the register. This means the interface has to allow setting callbacks for core read and write for all I/O registers associated with the peripheral.
(3) **General Tasks**: While the peripheral logic can run in a separate thread, it might be necessary to perform functionality with fine-grained timing control. To this end, the peripheral should implement a callback where the emulator state (including the number of passed cycles) is available.

Given these parts have been implemented, a peripheral module still has to be combined with the emulation manager, assigning interrupt vectors for the specific peripheral and I/O register callbacks to the correct register addresses for the MCU being used.

5.2 Instruction Decode

The first step in emulating an AVR MCU is implementing instruction decode and decoding the AVR ISA can be messy to do in software. Operands may be unevenly split across byte or nibble boundaries and special cases make writing a decoder error-prone and inefficient. A reasonable and fast approach is to use a table-based decoder. Because almost all instructions are exactly two bytes long and wide instructions can be discerned based on the first two bytes, every instruction can be decoded using two table lookups, one for the high and one for the low byte. However, this comes with a large amount of manual code duplication.

While the table-based decoding approach should be in general preferred, AVRS implemented a smaller, hand-written instruction parser. While all other emulators perform the decode and the emulation of the instruction at the same time, AVRS decodes the whole flash memory and stores an intermediate representation (IR) when first loading the firmware. While this opens the possibility for conducting binary analysis techniques on the IR itself, there are additional benefits.

The AVRS IR, represented as typed Rust enumeration, was designed in order for the actual emulation loop having to do as little work as possible when fetching the IR instruction. An example for this is the resolving of relative jumps: AVRS IR instructions only store absolute addresses and relative jumps are resolved during decoding. As an absolute address pointing to flash memory can have a maximum size of 24-bit, the address is split up into a 16-bit lower and an 8-bit higher part. Due to Rust having to store the enumeration type tag in the first byte of the IR in memory, splitting the 24-bit address up instead of just using a 32-bit integer, allows IR instructions to fit within 4 bytes, meaning the IR will take up at most twice the amount of flash memory.

5.3 Instruction Emulation

When actually executing the emulation loop, the IR brings an additional feature: when looking at the disassembled code of AVRS, one can see that executing the IR optimizes to a jump table, which boosts the core performance (see subsection 6.4). Despite already having an intermediate representation, some instructions behave differently depending on the core they are executed on.

One of these cases is accessing memory through pointer registers, as the effect of this instruction depends on the size of SRAM available to the core. If there is less than 256 bytes of SRAM available, only the lower byte of the pointer register is being used for memory access. With the existence of less than 65536 bytes of SRAM, both bytes of the pointer register are being used for the memory access, and if even more SRAM is available, the RAMP registers are being used. A different example is the size of the program counter being dependent on the amount of flash memory being present in the device. For easier handling of certain instructions, the status register and the stack pointer are not stored and retrieved from actual I/O memory, but only stored as an emulator internal state. The MMU handler intercepts memory access to these locations and updates the emulator internal state accordingly.

5.4 Peripheral Communication

The MMU handler is the first step in interfacing with peripherals, as it checks whether the memory access concerns I/O mapped memory. If an I/O register has been accessed, the MMU cannot fulfill the request on its own and passes it to the emulation manager. The emulation manager then uses a lookup table to decide which I/O callback is triggered, passing control to the peripheral implementation.

Handling of peripheral state change is expected to run in threads separated from the main emulation thread. By using Rust channels in I/O callbacks, the memory read/write information is communicated with the peripheral thread for mimicking transmission behavior, e.g. UART transmissions. As I/O mapped memory typically triggers side-effects, listening on the I/O channels in the peripheral thread allows asynchronous handling of these side-effects, while the emulation loop continues to run. As registering a channel for every location in data memory would cause too big of an overhead, all memory read/writes are logged onto a tracking channel, if the channel has been opened by an external caller. As keeping track of all channels can become quite bothersome, Rust macros are leveraged to generate code that would have to be manually duplicated otherwise.

6 EVALUATION

In order to establish a fair comparison, the emulators are being rudimentarily tested for correct functionality, with AVRS undergoing more rigorous testing. Afterwards, the feature set implemented by the different emulators, the supported instructions of the AVR ISA and finally, the performance on a set of example programs is being compared.

6.1 Validation and Test Programs

Testing the AVRS decoder is done by comparing the internal representation used by AVRS to the output of *objdump* and *radare2* [5], where *avr-objdump* is considered the ground truth. Rather than testing edge cases for single instructions, the whole 16-bit instruction space is enumerated, where the second half of two-word instructions is filled with the constant *0xffff*. Because some cores of the Tiny family use a reduced instruction set with overlapping instructions, the instruction space has to be enumerated again for these instructions.

Decoding and disassembly of instructions are tested separately, as the AVRS internal representation does not always represent the textual disassembly output. After disassembling the enumerated instruction space with *avr-objdump*, a script is being used to transform the disassembly into the internal AVRS representation. Using the opcode and the internal representation, Rust test cases are generated for every instruction. For testing the correctness of the disassembly, the target disassembly is generated by using *radare2* [5] and then compared to the output by AVRS; AVRS passes both tests. In the course of testing AVRS instruction decode, a bug in *avr-objdump* has been identified: when disassembling *LDS* or *STS* instructions of the Tiny family, the wrong disassembly would be displayed. The bug has been reported and since been fixed [6].

Testing the instruction semantics requires more manual effort, as generating test cases from reference output is not possible. Simavr and Avrora already provide test cases for implementations of their instructions, which have been ported. Adding custom test cases is still necessary for instructions not covered by Avrora and Simulavr. In order to test all emulators for basic functionality, a set of test programs, involving basic functionality, such as blinking LEDs and UART transmission, have been executed and judged by their behavior in the emulators, with the results being listed in Table 1. Only one test case did not pass, as atemu does not appear to provide any facilities for EEPROM handling; the table also lists, whether there has been an implementation of a peripheral, providing feedback on the peripheral state, such as printing UART transmissions to standard output, but a test program is still considered working when the correct memory access to the peripheral specific region is being made.

Program	atemu	Avrora	AVRS	GNU	IMAVR	simavr	SimulAVR
BLINK	●	●	●	●	◐	●	●
UART_RX	◐	●	●	◐	●	●	●
UART_RXINT	◐	●	●	◐	●	●	●
UART_TX	●	●	●	◐	●	●	●
UART_TXINT	●	●	●	◐	●	●	●
UART_ECHO	◐	●	●	◐	●	●	●
EEPROM	○	●	●	●	●	●	●

● = working with peripheral feedback, ◐ = working, ○ = not working

Table 1: Support of test programs across all emulators.

Instruction	atemu	Avrora	AVRS	GNU	IMAVR	simavr	SimulAVR
BREAK	✓	✓	✓			✓	✓
DES			✓				
EICALL	✓		✓			✓	✓
EIJMP	✓		✓			✓	✓
FMUL	✓	✓	✓	✓		✓	✓
FMULS	✓	✓	✓	✓		✓	✓
FMULSU	✓	✓	✓	✓		✓	✓
LAC			✓				
LAS			✓				
LAT			✓				
LDS_TINY			✓				
SLEEP	✓	✓	✓	✓	✓	✓	
SPM	✓	✓	✓			✓	✓
SPM2			✓				✓
STS_TINY			✓				
WDR	✓		✓	✓		✓	✓
XCH			✓				

Table 2: Listing of implemented instructions, without instructions that have been implemented by all emulators.

6.2 Comparison of Capabilities

A quantified overview of emulator capabilities we are interested in can be found in Table 3. Out of the pre-existing emulators, no emulator is a perfect fit, be it differing support for various file formats, missing crucial device families or lacking debug functionalities. SimulAVR and simavr are shown as most mature emulators, bringing a large number of implemented cores and peripherals.

6.3 Comparison of Implemented Instructions

Through manual source code analysis of each emulator, all implemented instructions have been listed, with instructions implemented by all emulators being stripped from the listing. The results are presented as a table in Table 2, showing a clear pattern of lacking support for the 16-bit LDS/STS instructions, as well as rarely used XMega instructions, such as *DES* or *LAC*. The table only credits the presence of the instruction in the code, which does not necessarily imply correctness. When confronted with large amounts of memory, not all emulators implement the RAMP registers for memory page selection, or instructions which require processor support state, such as the watchdog reset. Behavior of such instructions can then be different across emulators, e.g. the *BREAK* instruction does not halt execution on every emulator, as GNU AVR and IMAVR do not recognize the instruction, with GNU AVR failing to even load the firmware containing it.

6.4 Comparison of Performance

All performance measurements have been conducted on a Ubuntu 18.04.4 LTS VM on a server running an Intel®Xeon®CPU E5-2630 v4 at 2.20GHz. Resources available to the VM were 4 logical processors and 4 GB of RAM. All emulators, except AVRS, have been compiled using the toolchains available on this Ubuntu version, AVRS has been compiled using stable *rustc 1.40.0 (73528e339 2019-12-16).*

To measure performance and correctness in a fair way, the tested emulators have been modified to print a timer output, to terminate when reaching a *break* instruction and to print a memory dump of the data memory on termination; these changes have been made using the least invasive method available for each emulator. During execution, outputs are redirected to */dev/null* and our debug outputs printed to *stderr*.

A script executes the benchmarks across all emulators, collecting the printed timers in a CSV file, including the baseline performance. The Atmel Studio Emulator [25] has been used to determine the number of cycles executed, picking the ATmega128 as core with a frequency of 16 MHz. Every performance measurement for all benchmarks is repeated 25 times for each emulator.

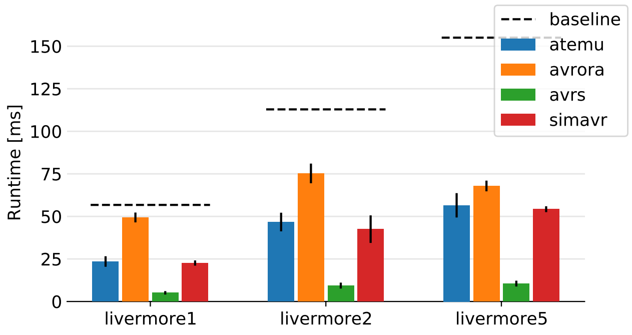

Figure 3: Measuring Livermore loop program runtime.

While making the performance counter modifications to the emulators, it became clear that it is not possible to include the GNU AVR Emulator in the performance analysis. Due to the fact that the emulator code is tightly coupled with the XWindow GUI code there is no way to add performance counters without rewriting large parts of the emulator. A similar situation caused the exclusion of IMAVR, which produced invalid results when compared with the reference memory dump.

The remaining five emulators have then been profiled using Livermore loops, repeated AES encryption, public key cryptography and hashing; Livermore loops have been included for comparison with benchmarks mentioned in the Avrora publication [43]. The goal of these benchmarks was to show raw instruction emulation performance, as peripheral performance cannot be measured objectively across emulators.

Figure 3 shows that the gap between Avrora/atemu and SimulAVR has widened in comparison to the older results. Because SimulAVR is slower than the other emulators by an order of magnitude and the only emulator to be slower then the physical ATmega128 baseline, it has been removed from the benchmark graphs to highlight the difference between the other emulators.

Emulator	Input	Cores	Core Families	Peripherals	Debugging	OS	GUI
Avrora	ELF, Disassembly	3	Mega	EEPROM, UART, I/O ports, Timers, Radio, SPI, ADC	Tracing, GDB-Server	Windows, Linux, OS X	No
simavr	IHEX	28	Mega, Tiny	EEPROM, Watchdog, UART, I/O ports, Timers, SPI, ADC, I^2C	GDB-Server	Windows, Linux, OS X	Peripherals
SimulAVR	ELF	33	Mega, Tiny	EEPROM, Watchdog, UART SPI, ADC, USI, I/O Ports	GDB-Server	Windows, Linux	Peripherals
atemu	ELF, SREC	1	Mega	Radio, Timers, UART SPI, ADC, I/O Ports	Debug GUI	Linux	Minimal
GNU AVR Simulator	IHEX, SREC	22	Mega	EEPROM	Debug GUI	Linux	Minimal
IMAVR	Binary	5	Mega	Timers, UART, JTAG, I/O Ports	Debug Shell	Linux	No
AVRS	IHEX	3	Tiny, Mega, XMega	UART, EEPROM, I/O Ports	Debug GUI	Windows, Linux, OS X	Yes

Table 3: Overview of emulator capabilities.

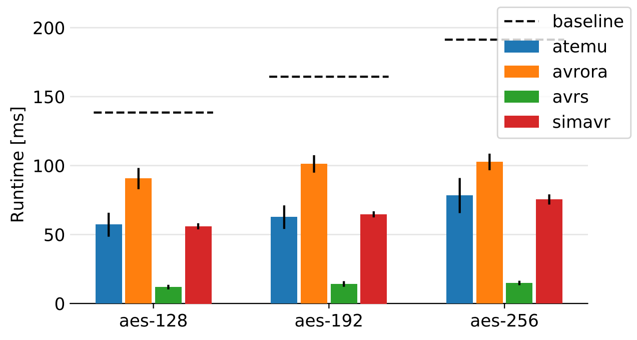

Figure 4: Measuring runtime of repeated AES encryption.

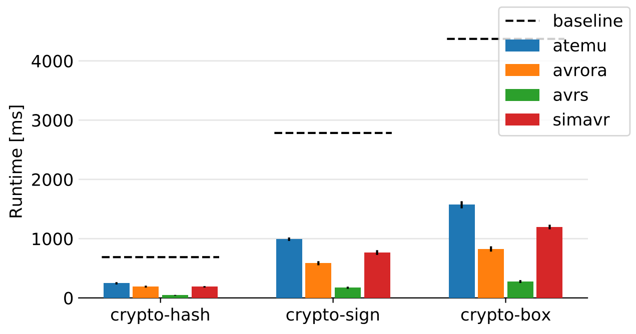

Figure 5: Measuring runtime of repeated hashing, ed25519 signatures and curve25519 public key encryption.

When comparing the faster emulators, Avrora shows the worst performance on the Livermore loops, while atemu and simavr show similar runtimes. However, AVRS performs best, being about four times faster compared to atemu and simavr. The same observations can be made on the AES benchmarks in Figure 4, with one notable difference: while the runtime increases on the physical device and for SimulAVR, the runtime of the other emulators does not change by a large margin, with the step from *aes-128* to *aes-192* being barely noticeable.

The results of the hashing/public key cryptography benchmark in Figure 5 using AVRNaCl, apply heavier computational load on the devices than the AES implementation does. Avrora profits from this type of load and can achieve better runtime measurements than atemu and simavr, while still being slower than AVRS. While only being about three times faster than Avrora, AVRS also performs the best on the AVRNaCL benchmark.

As a nod to the original Avrora benchmarks, Figure 6 provides an overview of the frequencies that have been achieved during the execution of the benchmarks. Every benchmark result has been converted into frequencies by referring to the baseline runtime and frequency, and the mean and standard deviation have been plotted in the figure. Due to Avrora's performance on the AVRNaCL benchmark, the achieved mean frequencies of Avrora, atemu and simavr are all close to 50 MHz, while SimulAVR performance did not improve since the original Avrora benchmarks. However, AVRS manages to run at the highest measured frequency. While AVRS' frequency distribution is more scattered, it still allows driving the simulation with more than 200 MHz in most cases. This benchmark is intended to show the theoretically achievable frequencies; the logical frequency used to simulate cycle accuracy will always match that of a real physical device instead.

There are two deciding factors in AVRS' performance in comparison to other emulators. For one, decoding the instructions ahead of the actual emulation is the deciding factor in performance. While this allows decoding ahead of the instruction emulation loop, the IR used by AVRS causes the emulation loop to be constructed as a jump table. Secondly, AVRS relies heavily on Rust macros and code duplication during compile time. This is a tradeoff, as it substantially increases compile time and output binary size with each added core, while being able to hardcode and optimize core-specific information at compile time, in comparison to other emulators, where core differences are handled during runtime.

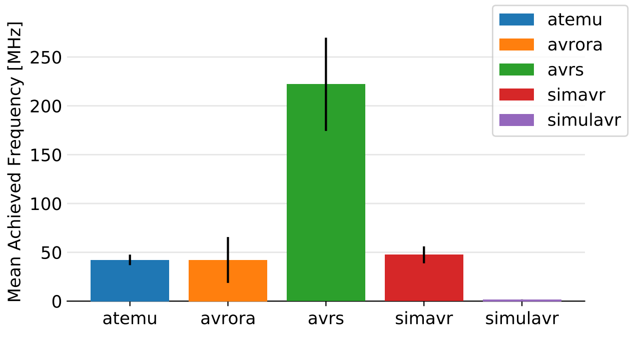

Figure 6: Mean achieved frequencies per emulator, calculated from all gathered performance results.

7 FUZZING

In order to show the usefulness of AVRS as a tool for vulnerability discovery, the ability to fuzz firmware has been added on top of it. A fuzzer generates inputs and runs a target program on these inputs, with the goal of triggering a vulnerability in the program. Since fuzzing firmware on embedded devices is different to fuzzing application software, as shown by Muench et al. [28], we are relying on their work for detection of crashes.

7.1 Setup

AVRS marks the base of the fuzzing setup and is first compiled for the target core, including the required peripherals of the board to be fuzzed. Generating the fuzzing inputs is handled by *boofuzz* [29], which has been chosen for it's serial support and to mirror the setup used by Muench et al. [28]. The fuzzer itself does not contain any instrumentation logic for introspection of the emulator state; this is handled on the emulator side, by logging state from the top-level emulation loop or by making use of peripherals. Whether a crash occurred is determined by the emulator itself as well, either by triggering Rust runtime checks, e.g. out-of-bounds memory access, or by detecting crash conditions and manually aborting the emulator. Both cases result in a termination of the emulator with an error return code, which is detected by a *boofuzz* program monitor.

7.2 Heuristics

Muench et al. [28] demonstrated the difficulties of fuzzing embedded devices, where memory corruption might not result in observable crashes of the program, compared to desktop applications. They solved this problem by introducing heuristics which would trigger on certain cases of memory corruption. We picked a subset of these heuristics to implement in AVRS, choosing the ones appropriate for even the smallest AVR cores without dedicated operating systems. These heuristics are added to the emulator code and compiled into the board to be fuzzed, where we utilize Rust panics to terminate the emulator, and let the fuzzer identify the crash. As there might still be memory corruptions that are not caught by the emulator, a liveness check is required. This check makes sure the firmware performs a certain action as expected. It has to be implemented as an extension to the fuzzer and is dependent on the application to be tested. Alternatively the fuzzer would need to employ timeouts to re-start fuzzing.

Segment Tracking. As there is no explicit memory segmentation in the AVR architecture, memory reads and writes cannot be directly classified as invalid, as even a null-pointer dereference is in general a valid and common memory access. However, some aspects of this heuristic can still be reused, albeit specifically tailored to one AVR core and/or one firmware program. Revisiting Figure 1 shows that there are reserved memory areas between different peripherals mapped in the data memory. If a memory access happens in such an area, it can be classified as invalid. In the simplest case, this means a memory access beyond the highest available data memory address is always invalid, which is directly caught by Rust as out-of-bounds access. Memory access can also be constrained on a per-instruction basis: AVRS implements this by storing an additional 8-byte integer for every instruction, setting a read and write bit for up to 32 sections in data memory. These sections are intended to be defined according to the datasheet and allow specific parts of the firmware to only access certain sections in memory; e.g. a routine performing calculations in SRAM does not need memory access to the peripheral section of data memory. This simple method effectively triples the amount of memory needed for the firmware, which is still negligible, due to small AVR flash sizes.

Format Specifier Tracking. To detect misuse of format strings, the implemented heuristic [28, 39] relies on detecting the presence of the format string in the binary and whether the address of the format string argument is located in a read-only section; as there are again no memory segments with permission bits, this heuristic cannot be implemented reliably. However, it can be tracked whether memory contents have been copied from flash memory to SRAM unchanged: while this depends on the used compiler, avr-gcc usually generates code copying constant data to SRAM right before the main program is being called, allowing this memory to be tagged read-only until further modifications are made. This is extended in a more general notion, as we track all program- to data-memory copies, by linking a program memory load with the next data memory write if their contents are equal and tagging them again as read-only. AVRS can then be supplied with the addresses of critical format specifier functions and the location of the first argument according to calling convention, and terminate the emulation if the format string contains bytes which are not tagged read-only. This heuristic can be reused in application specific contexts, where critical functions should only accept read-only data.

Call Stack/Frame Tracking. Using a shadow stack [47] built into AVRS, every call and return instruction is tracked. On every call, the intended return address is stored in the shadow stack and compared with the actual return address on return, which also works for interrupts in AVRS. Not being able to handle interrupts with this technique was cited as a source of false negatives in [28], but due to the semantics of interrupts on AVR, which handle interrupts in a similar manner to function calls, this is not an issue here. However, peculiarities in generated code can result in false positives, with one example shown in Listing 1. In this case, a generated function *fcn_24c* only performs minimal actions and immediately calls another function; the callee subsequently skips the stack frame of *fcn_24c* on return altogether.

In order to mitigate this, AVRS also tracks *PUSH* and *POP* instructions on the shadow stack, in case a stack frame is eliminated this way. This does not solve the issue in Listing 1, as the return address is removed by direct manipulation of the stack pointer. Instead of allowing this behavior with the shadow stack, we decided to leverage the segment tracking approach for this, in order to manually resolve false positives to reduce risk of false negatives. AVRS also tracks call frames, as suggested in [28], but this comes with similar constraints as return address tracking: the low amount of available SRAM encourages writes across stack frames to save memory, this introduces more false positives, which need to be manually excluded.

Custom Heuristics. As only heuristics which required additional features to be implemented in AVRS have been picked, the list of heuristics is not exhaustive and there are other heuristics, such as heap object tracking [38]. However, these heuristics could be added using the AVRS peripheral interface, as it provides the means to track and intercept memory access.

7.3 Examples

To put the implemented heuristics to use, we prepared two examples utilizing serial communication, on two different cores: ATtiny104, due to the only MCU of the Tiny family featuring the reduced instruction set and a UART port, and the ATmega328P, due to its popularity within the Arduino project. The firmware programs, implementing protocols over UART have been implanted with intentional crafted vulnerabilities, but use different features on the different cores, as the ATtiny104 does not provide enough SRAM to use format strings or heap allocation in a reasonable manner, implying that there are no format strings and heap vulnerabilities on the ATtiny104 firmware. To start the fuzzing loop, a *boofuzz* script is provided for both cores, with a specific liveness check being implemented for both cores. Beyond covering unresponsiveness of the program due to vulnerabilities, this check ensures an emulator restart in case of issues with *boofuzz*' serial connection.

Running the scripts results in *boofuzz* being able to repeatedly find inputs triggering the implanted vulnerabilities, with the exception of implanted heap vulnerabilities. This can be mitigated to some degree by employing segment tracking for coarse-grained out-of-bounds write detection, but it can not replace a dedicated heap-object tracking mechanism. As the fuzzing setup is being published alongside the AVRS source code, the fuzzing core for ATmega328P enables libraries designed for usage within Arduino-based projects to be analyzed, when the needed peripherals can be implemented.

8 CONCLUSION

In this paper we discussed the challenges of emulating microcontrollers of the AVR architecture. We explored the landscape of existing AVR emulators, pointing out missing architectural features and shortcomings in the extensibility of emulators. While there are mature and actively developed emulators, such as simavr, none of them could fulfill our requests with respect to observability of internal emulation events and feature completeness. This lack manifested in the missing support for the smallest and most advanced AVR cores respectively, the Tiny and XMega families. To improve this situation, we implemented AVRS, a new emulator designed to provide completeness of general AVR features and an emulation pipeline which can be intercepted at the necessary granularity to build tools for security analysis. We introduced an intermediate representation of AVR opcodes in AVRS, allowing us to split instruction decode from emulation and potentially enabling static analysis on the IR. This implementation detail was reflected in the comparison against existing emulators, where AVRS displays competitive performance alongside feature completeness. Finally, we built a fuzzing tool on top of AVRS, using the heuristics and techniques described in [28].

```
.fcn_24c:
  push r28
  push r29
  rcall fcn_252

.fcn_252:
  in r28, 0x3d      ; SPL
  in r29, 0x3e      ; SPH

  ; ... function epilogue
  adiw r28, 0x03    ; Skip return address
  out 0x3d, r28     ; SPL
  out 0x3e, r29     ; SPH
  pop r29
  pop r28
  ret
```

Listing 1: Epilogue accessing stack of previous callframe.

8.1 Future Work

AVRS is designed as a base for tools to build upon and given a plethora of existing tools in the field of dynamic analysis, the next logical step is integration of AVRS into these frameworks where an emulator is needed. Examples for this are the PANDA platform [15] or the Avatar^2 framework [27]; the latter opens the question if it is possible to forward peripheral interaction in the same manner as on other platforms, which eliminates the need for peripheral emulation to some degree. Other possibilities include the facilitation of concolic execution by combining AVRS with e.g. Angr [41] to improve coverage in the presented fuzzer. Lastly, we believe that AVRS can be used to study new dynamic analysis variants for embedded systems on a comparatively small architecture, before applying them to more capable microcontrollers.

ACKNOWLEDGMENTS

This research was funded by the FFG Bridge 1 871230 iSaFe and FFG Bridge Young Scientists 871230 AutoHoney grants, as well as the competence center SBA Research (SBA-K1) funded by COMET (FFG 871230).

REFERENCES

[1] 2018. CVE-2018-17614. Available from MITRE, CVE-ID CVE-2018-17614.. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-17614

[2] National Security Agency. 2019. Ghidra. https://ghidra-sre.org/

[3] Carlos Alberca, Sergio Pastrana, Guillermo Suarez-Tangil, and Paolo Palmieri. 2016. Security analysis and exploitation of arduino devices in the internet of things. In *Proceedings of the ACM International Conference on Computing Frontiers*. 437–442.

[4] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. 2019. SoK: Security Evaluation of Home-Based IoT Deployments. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1362–1380. https://doi.org/10.1109/SP.2019.00013

[5] Sergi Alvarez. 2006. radare2: Libre and Portable Reverse Engineering Framework. https://www.radare.org/n/

[6] binutils Bugzilla. 2019. Invalid disassembly of avrtiny LDS/STS instructions. https://sourceware.org/bugzilla/show_bug.cgi?id=25041. accessed 2020-01-31.

[7] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. 2020. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. *arXiv preprint arXiv:2002.03416* (2020).

[8] Andrew Carter. 2016. micro-aes - A permissively licensed AES implementation optimised for running on micro-controllers. https://github.com/SmarterDM/micro-aes. accessed 2019-10-04.

[9] Z Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick McDaniel. 2019. Program analysis of commodity IoT applications for security and privacy: Challenges and opportunities. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–30.

[10] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware.. In *NDSS*, Vol. 16. 1–16.

[11] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.. In *NDSS*.

[12] Atmel Corporation. 2014. ATxmega16A4U/32A4U/64A4U/128A4U Datasheet. http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8387-8-and16-bit-AVR-Microcontroller-XMEGA-A4U_Datasheet.pdf. accessed 2020-01-25.

[13] Atmel Corporation. 2016. AVR Instruction Set Manual. http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf. accessed 2019-07-26.

[14] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2016. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 437–448.

[15] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. 2013. Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 839–850. https://doi.org/10.1145/2508859.2516697

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

[17] Sergey Gulchuck. 2013. IMAVR Project Website. http://imavr.sourceforge.net/. accessed 2019-08-26.

[18] Michael Hutter and Peter Schwabe. 2013. NaCl on 8-bit AVR Microcontrollers. In *Progress in Cryptology – AFRICACRYPT 2013 (Lecture Notes in Computer Science, Vol. 7918)*, Amr Youssef and Abderrahmane Nitaj (Eds.). Springer-Verlag Berlin Heidelberg, 156–172. Document ID: cd4aad485407c33ece17e509622eb554, http://cryptojedi.org/papers/#avrnacl.

[19] Microchip Technology Inc. 2018. Atmel Studio 7 Production Files. https://www.microchip.com/webdoc/GUID-ECD8A826-B1DA-44FC-BE0B-5A53418A47BD/index.html?GUID-E9FD4617-290B-4EA2-8C82-B1524094A495. accessed 2019-08-27.

[20] Intel. 1988. Hexadecimal Object File Format Specification (Revision A). https://archive.org/details/IntelHEXStandard. accessed 2020-01-20.

[21] Timo Kerstan and Markus Oertel. 2010. Design of a real-time optimized emulation method. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE. https://doi.org/10.1109/date.2010.5457126

[22] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. 2003. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems* (Los Angeles, California, USA) *(SenSys '03)*. ACM, New York, NY, USA, 126–137. https://doi.org/10.1145/958491.958506

[23] Amit Mandal, Pietro Ferrara, Yuliy Khlyebnikov, Agostino Cortesi, and Fausto Spoto. 2020. Cross-program taint analysis for IoT systems. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 1944–1952.

[24] Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi. 2010. *AVR Microcontroller and Embedded Systems: Using Assembly and C* (1st ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.

[25] Microchip Technology Inc. 2018. Atmel Studio 7. https://www.microchip.com/mplab/avr-support/atmel-studio-7. accessed 2019-08-19.

[26] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, Vol. 18. 1–11.

[27] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar^2: A multi-target orchestration platform. In *BAR 2018, Workshop on Binary Analysis Research, colocated with NDSS Symposium, 18 February 2018, San Diego, USA*. San Diego, UNITED STATES. http://www.eurecom.fr/publication/5437

[28] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA*. San Diego, UNITED STATES. http://www.eurecom.fr/publication/5417

[29] Joshua Pereyda. 2016. boofuzz. https://github.com/jtpereyda. accessed 2020-04-21.

[30] Isaías González Pérez, Antonio José Calderón Godoy, Manuel Calderón Godoy, and Juan Félix González González. 2019. Survey about the Utilization of Open Source Arduino for Control and Measurement Systems in Advanced Scenarios. Application to Smart Micro-Grid and Its Digital Replica. In *ICINCO*.

[31] Tim Peters. 1992. Livermore loops coded in C. https://www.netlib.org/benchmark/livermorec. accessed 2019-10-04.

[32] Michel Pollet and Jakob Gruber. 2019. simavr Code Repository. https://github.com/buserror/simavr. accessed 2019-10-03.

[33] J. Polley, D. Blazakis, J. McGee, Dan Rusk, J.S. Baras, and M. Karir. 2004. ATEMU: a fine-grained sensor network simulator. In *2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004*. IEEE. https://doi.org/10.1109/sahcn.2004.1381912

[34] J. Polley, D. Blazakis, J. McGee, Dan Rusk, J.S. Baras, and M. Karir. 2004. atemu Project Website. http://www.hynet.umd.edu/research/atemu/. accessed 2019-01-20.

[35] Michael Rolnik. 2020. QEMU AVR Patch Set. https://patchew.org/QEMU/20200118191416.19934-1-mrolnik@gmail.com/. accessed 2020-07-04.

[36] Theodore Roth and Klaus Rudolph. 2012. SimulAVR Project Website. https://www.nongnu.org/simulavr/. accessed 2019-10-03.

[37] Patrick Schaumont, Patrick Schaumont, Doris Ching, and Ingrid Verbauwhede. 2006. An Interactive Codesign Environment for Domain-specific Coprocessors. *ACM Trans. Des. Autom. Electron. Syst.* 11, 1 (Jan. 2006), 70–87. https://doi.org/10.1145/1124713.1124719

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[39] Team Shellphish. 2017. Cyber Grand Shellpish. Phrack Papers. http://www.phrack.org/papers/cyber_grand_shellphish.html

[40] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware.. In *NDSS*.

[41] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[42] QEMU Team. 2020. QEMU: the FAST! processor emulator. https://www.qemu.org/. accessed 2020-07-04.

[43] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. 2005. Avrora: Scalable Sensor Network Simulation with Precise Timing. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks* (Los Angeles, California) *(IPSN '05)*. IEEE Press, Piscataway, NJ, USA, Article 67. http://dl.acm.org/citation.cfm?id=1147685.1147768

[44] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. 2011. Avrora Project Website. http://compilers.cs.ucla.edu/avrora/. accessed 2019-10-03.

[45] Ben L. Titzer and Jens Palsberg. 2005. Nonintrusive Precision Instrumentation of Microcontroller Software. *SIGPLAN Not.* 40, 7 (June 2005), 59–68. https://doi.org/10.1145/1070891.1065919

[46] Sergiy Uvarov. 2002. GNU AVR Simulator Project Website. https://sourceforge.net/projects/avr/. accessed 2019-01-20.

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

[48] Andrew Wickert, Chad Sandell, Bobby Schulz, and G.-H Ng. 2018. Open-source Arduino-derived data loggers designed for field research. *Hydrology and Earth System Sciences Discussions* (12 2018), 1–16. https://doi.org/10.5194/hess-2018-591

[49] Jingyao Zhang, Yi Tang, Sachin Hirve, Srikrishna Iyer, Patrick Schaumont, and Yaling Yang. 2011. A software-hardware emulator for sensor networks. In *2011 8th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*. IEEE. https://doi.org/10.1109/sahcn.2011.5984928

[50] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1099–1114.