ICICLE: A Re-designed Emulator for Grey-Box Firmware Fuzzing

Michael Chesser
The University of Adelaide
Data61 CSIRO, Cyber Security Cooperative Research Centre
Australia
michael.chesser@adelaide.edu.au

Surya Nepal
Data61 CSIRO, Cyber Security Cooperative Research Centre
Australia
Surya.Nepal@data61.csiro.au

Damith C. Ranasinghe
The University of Adelaide
The School of Computer Science
Australia
damith.ranasinghe@adelaide.edu.au

ABSTRACT

Emulation-based fuzzers enable testing binaries without source code and facilitate testing embedded applications where automated execution on the target hardware architecture is difficult and slow. The instrumentation techniques added to extract feedback and guide input mutations towards generating effective test cases is at the core of modern fuzzers. But, modern emulation-based fuzzers have evolved by re-purposing general-purpose emulators; consequently, developing and integrating fuzzing techniques, such as instrumentation methods, is difficult and often added in an ad-hoc manner, specific to an instruction set architecture (ISA). This limits state-of-the-art fuzzing techniques to a few ISAs such as x86/x86-64 or ARM/AArch64; a significant problem for *firmware fuzzing* of diverse ISAs.

This study presents our efforts to *re-think emulation for fuzzing*. We design and implement a fuzzing-specific, multi-architecture emulation framework—ICICLE. We demonstrate the capability to add instrumentation once, in an *architecture agnostic* manner, with low execution overhead. We employ ICICLE as the emulator for a state-of-the-art ARM firmware fuzzer—FUZZWARE—and replicate results. Significantly, we demonstrate the availability of new instrumentation in ICICLE enabled the discovery of new bugs. We demonstrate the fidelity of ICICLE and efficacy of architecture agnostic instrumentation by discovering bugs in benchmarks that require a known and *specific* operational capability of instrumentation techniques, *across a diverse set of instruction set architectures* (x86, ARM/AArch64, RISC-V, MIPS). Further, to demonstrate the effectiveness of ICICLE to discover bugs in a currently *unsupported* architecture in emulation-based fuzzers, we perform a fuzzing campaign with real-world firmware binaries for Texas Instruments' MSP430 ISA and discovered 7 new bugs.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Security and privacy → Embedded systems security**.

KEYWORDS

Fuzzing, emulation, embedded systems

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598039

ACM Reference Format:
Michael Chesser, Surya Nepal, and Damith C. Ranasinghe. 2023. ICICLE: A Re-designed Emulator for Grey-Box Firmware Fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598039

1 INTRODUCTION

Fuzzing is an automated software testing methodology that repeatedly executes a program with generated inputs and monitors execution for adverse behaviors. Progress in the field has greatly enhanced the *bug discovery* capability of modern fuzzers and fuzzing is now widely used in the software development industry. In particular, grey-box fuzzing (or feedback-driven) methods have proven to be highly effective at scale and are capable of finding bugs in a diverse set of software [3, 4, 7, 33, 40, 46, 52, 55]. Grey-box fuzzing relies on the ability to add instrumentation to the target program to obtain feedback. This feedback allows input generation to be intelligently guided, improving a fuzzer's ability to discover bugs.

A simple method to facilitate grey-box fuzzing is for the compiler to inject instrumentation into the source code during compilation. However, it is often necessary to fuzz binaries where source code is unavailable—*binary-only fuzzing*—or where the target hardware is not suitable for automating testing and testing is carried out on a host machine with a different instruction set architecture (ISA)—*cross-architecture fuzzing*. For instance, it is extremely challenging to perform rapid execution on devices typically used for Internet of Things (IoT) applications and embedded systems in general [11, 15, 24, 31, 32, 36, 41, 47, 68]. Consequently, we are forced to use *emulators* capable of executing binaries built for the target on a more convenient host machine; exploiting the resource capabilities of the host for software bug discovery and triaging. Therefore, emulators play a critical role in supporting both *binary-only* and *cross-architecture* fuzzing. Significantly, emulators enable unparalleled control and introspection over program execution, even without source code and access to the original hardware.

Current state-of-practice for emulation-based grey-box fuzzing, driven by its more recent evolution compared to emulators, is to integrate fuzzing instrumentation into existing general-purpose emulators. But, this can be challenging because these emulators were not designed to support such modifications [53]. Consequently, existing emulation-based fuzzers implement instrumentation techniques either manually, through direct modifications to the emulator [26, 27, 53, 66], or through limited interfaces that are unable to support more advanced instrumentation [31, 41, 42, 51]. Further, the absence of a consistent approach to add new, experimental,

instrumentation and the need for domain expertise in emulator development to evaluate new fuzzing techniques are arguably barriers to developments in the field. As a result, the benefits of extensive research efforts to develop state-of-the-art fuzzing techniques can remain limited to a specific ISA; this is *undesirable*.

Our Contributions. This study presents our efforts to design and build a *new* multi-architecture emulation framework explicitly for fuzzing.

In summary, we make the following contributions:

- We designed a new multi-architecture emulator, Icicle, for directly supporting emulation-based fuzzing: i) enabling the implementation of architecture-agnostic instrumentation; ii) employing a decoupled design, enabling emulation, instrumentation and instruction set architecture (ISA) support to be maintained separately; and iii) byte level memory-management to better support emulating memory in embedded systems.
- We implemented Icicle as a coverage-guided greybox fuzzer by integrating with AFL++ and Fuzzware.
- We conducted extensive experiments across five diverse ISAs, multiple instrumentation techniques, 21 real-world binaries and a synthetic test program. Notably, we *demonstrate*: i) the instrumentation requirements of state-of-the-art fuzzing techniques are satisfied with a unified instrumentation interface without the need for architecture-specific knowledge; ii) the effectiveness of Icicle by comparing against existing emulators for the challenging task of firmware fuzzing. Significantly, Icicle, supported by additional architecture-agnostic instrumentation, uncovered previously undiscovered bugs; and iii) the efficacy of Icicle and its architecture agnostic instrumentation on a new ISA—we fuzz and discover seven bugs in real-world binaries written for Texas Instruments' MSP430 RISC architecture. Importantly, this ISA is currently *not supported* by existing emulation-based fuzzers.
- We ***open source***[1] Icicle to facilitate further improvements and advance the field of emulation-based fuzzing in general.

2 INSTRUMENTATION CHALLENGES IN EMULATION-BASED FUZZING

In this section, we present an overview of different instrumentation techniques that are used in modern grey-box fuzzing frameworks. Subsequently, we highlight the issues hindering their implementation in existing general-purpose emulators without resorting to direct, architecture-specific, modifications of the emulator that motivate the need of a re-designed emulator for fuzzing.

2.1 Instrumentation Techniques

Grey-box fuzzers rely on instrumentation techniques to obtain feedback to enable more effective exploration of a target program which is necessary for uncovering deeper bugs. Therefore, it is important for fuzzing frameworks to support a diverse set of instrumentation techniques. In this section we describe several instrumentation techniques developed in previous research efforts.

[1] https://github.com/icicle-emu/icicle

Code coverage (Branch hit counts). Almost all grey-box fuzzers utilize a form of code coverage for feedback. Code coverage identifies inputs that reach new locations within a program by instrumenting the target so it notifies the fuzzer when new code is reached. It is a proven and effective method that enables fuzzer to incrementally discover different parts of the program [8, 46]. Branch hit counts is an approach to code coverage popularized by AFL [66]. This approach maintains a global map of 8-bit counters that are incremented whenever an edge in the program is hit. After execution, the values of each of the counters grouped into one of 8 ranges and if any of the counters contain a value with a unique range, the fuzz input corresponding to the execution is considered novel.

Context-sensitive branch coverage [12, 64]. Context-sensitive branch coverage augments branch hit counts by hashing the edge index with the current calling context. This allows the fuzzer to obtain better feedback from branches inside of frequently called functions.

CmpLog [27]. For many target binaries, code coverage is insufficient at finding inputs that reach deep parts of the program. For example, comparisons against large constants (such as Listing 1) are difficult to satisfy with code coverage alone, because there is no feedback mechanism that allows for incremental progress towards solving the comparisons.

```
if (tag == 0x31677562) {
  crash();
}
```

Listing 1: Example program where `crash` is hard to reach with traditional code coverage instrumentation.

One approach explored in several recent studies [5, 25, 30] is to directly instrument the operands of comparisons. CmpLog is a comparison tracing technique implemented in AFL++ based on Redqueen [5] and Weizz [25]. CmpLog identifies comparisons within a program, then adds instrumentation that captures the operands of the comparison. After execution, the fuzzing frontend can then scan the input for the operands in order to replace them with their correct value (a process referred to as input-to-state replacement).

CompareCov [27]. Is an alternative approach to solving complex comparisons based an earlier compiler instrumentation technique, LAF-Intel [35]. CompareCov provides better feedback by instrumenting the program to split comparisons involving large values into comparison between individual bytes. The split comparison can subsequently be solved with code coverage instrumentation. In addition to instruction-level comparisons, CompareCov also attempts to improve feedback for memory comparison functions (`memcmp`, `strcmp` and `strncmp`) by adding instrumentation to update the coverage (bitmap) for every matching byte within the comparison operation.

2.2 Instrumentation Challenges

QEMU and Unicorn (forked from QEMU), have become the de facto emulators for fuzzers [14, 15, 17, 24, 31, 32, 41, 42, 47, 53, 54, 60, 61, 67–69]. Therefore, for the remainder of this paper we will primarily

compare ICICLE against emulator-level capabilities in QEMU and Unicorn to support fuzzing.

QEMU was primarily designed for fast, general-purpose, emulation, not for fuzzing. Consequently, many design decisions differ from those important for fuzzing. Crucially, it is difficult to add advanced instrumentation techniques in an architecture agnostic manner, for the following key reasons:

- QEMU's intermediate representation for dynamic translation, Tiny Code Generator (TCG) ops, is not designed for direct analysis or manipulation [53]. Consequently, it is challenging to add instrumentation without making invasive code changes at a very low level.
- QEMU is monolithic in design, providing little support for extensibility, and assumes that it controls the full life cycle of the emulation process. These properties inhibit the maintainability of modifications and reduce scalability as it makes it challenging to share state across fuzzing instances.
- Given QEMU's historical focus on emulation, there is no unified mechanism for adding instrumentation. For example, code coverage is implemented by modifying the code-generator to inject code at the start of every basic block; more complex techniques, such as CmpLog, modify the translation process of individual architectures. Hence, adding new instrumentation techniques require extensive emulation domain expertise.

Unicorn is a fork of QEMU designed for more flexible and modular CPU emulation. Unicorn extracts the CPU emulation component from QEMU, configures it to always use the software MMU implementation, and introduces hooks, functions that are called in response to selected emulator events, such as memory accesses and breakpoints [41, 42, 54].

Unicorn's function hooking API enables fuzzers to inject functions calls to observe the CPU state, which can be used to implement some instrumentation techniques. However, the observable state is architecture specific, and Unicorn provides no support for analyzing the code semantics, making more advanced instrumentation difficult. Additionally, the maintainers of the Unicorn project have also reported that it is difficult to keep Unicorn up-to-date with improvements because of QEMU's monolithic design [50].

Therefore, we are motivated to build a new multi-architecture emulation framework explicitly designed for fuzzing; with the ability to support sophisticated instrumentation methods in an architecture-agnostic manner to enable fast emulation-based fuzzing of binaries.

3 ICICLE DESIGN AND IMPLEMENTATION

We provide an overview of our fuzzing specific multi-architecture emulation framework in Figure 1. To enable architecture agnostic fuzzing we use a portable intermediate representation (IR) that

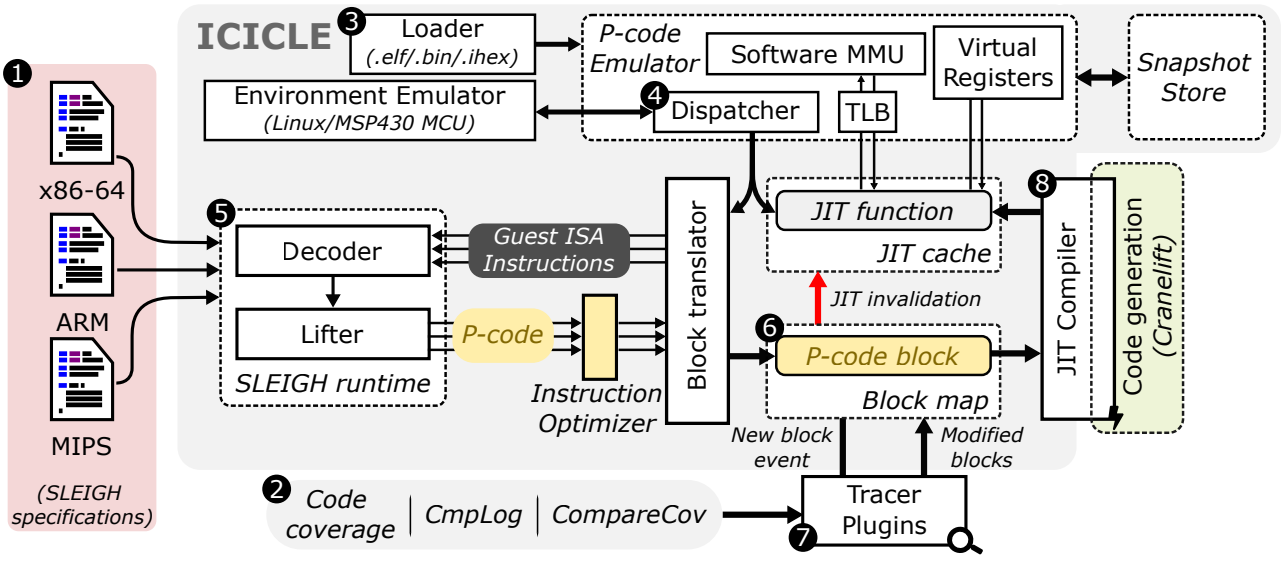

Figure 1: Overview of the core components in our fuzzing specific multi-architecture emulation framework. The instrumentation workflow is as follows: (1) On initialization the emulator loads the appropriate SLEIGH processor specification, configuring the SLEIGH runtime. (2) One or more *Tracer Plugins* are registered with the emulator to support the instrumentation needs of the fuzzer. (3) Once configured, the emulator loads the target binary into memory and starts execution. (4) During execution, whenever the emulator attempts to execute a new instruction, the dispatcher initiates the translation process. (5) Each guest instruction is then translated to P-code. (6) P-code operations are grouped into a block, and the block is stored in a global block map. (7) Tracer Plugins are notified to allow them to analyze the new block and modify any block in the block map required for instrumentation. (8) Any new or modified blocks are then compiled to native code using the JIT compiler and the emulator resumes execution. Notably, *all* of components within the grey area were implemented as part of ICICLE and only the SLEIGH specifications we employed were from Ghidra.

is suitable for both *emulation and program analysis*. Translation of the guest ISA to the portable IR is achieved using processor specifications that are external to the emulator. This ensures that architecture-specific details are kept decoupled; enabling fixes for specification bugs and the addition of new architectures to be implemented with minimal changes to the core emulator. Further, in contrast to existing emulation-based fuzzing frameworks, we define new instrumentation application programming interfaces (APIs) that enable instrumentation to exist entirely outside of the emulator. This facilitates researchers both, in developing new instrumentation techniques for emulation-based fuzzers without domain expertise in emulator development, and *the immediate availability of these techniques across ISAs* in a design-build-and-test *once-only* paradigm.

3.1 Fuzzing Specific Emulator Design

Icicle supports fast, multi-architecture, CPU emulation through portable dynamic translation. First, guest ISA instructions are translated to an intermediate representation (IR) called P-code. P-code is then just-in-time (JIT) compiled to the host architecture allowing for efficient execution.

Icicle performs translation to P-code through the use of SLEIGH[2] processor specifications for the guest ISA. SLEIGH is a domain-specific language (DSL) that describes how to decode and translate the semantics of machine code into P-code. We chose SLEIGH as the basis of Icicle's CPU emulation for the following reasons:

- *Broad architecture support.* We leverage the diverse set of SLEIGH processor specifications that have already been created as part of the open-source Ghidra framework [62] (over 45 processor kinds are supported). This enables Icicle to emulate a wide range of architectures, including architectures *unsupported* by other emulation frameworks like QEMU, with significantly reduced effort.
- *Designed for analysis.* Unlike IRs used in other emulator frameworks, P-code was explicitly designed to support program analysis. For example, P-code maintains hints for call and return operations even though such hints are unnecessary for emulation. This makes it better suited for performing the code analysis tasks required for advanced instrumentation techniques.
- *Suitable for emulation.* P-code consists of small set of operations that can be efficiently executed by the host ISA. For example, P-code avoids the use of bit-vectors. This allows for fast emulation.
- *Decoupling and ease of maintenance.* Icicle uses the original SLEIGH specifications written for Ghidra without any modifications. This allows any improvements or fixes made to a specification to be immediately usable by Icicle without any changes to the core emulator. Importantly, the use of SLEIGH specification naturally enables Icicle to emulate new targets by *simply* providing the new SLEIGH specification. Significantly, the new architecture will benefit from any existing instrumentation techniques *without* needed change to the emulator.

[2]The name SLEIGH was derived from SLED (Specification Language for Encoding and Decoding)[62], which also influenced the name of our emulator: Icicle

Icicle implements a P-code emulator[3] consisting of a SLEIGH runtime (a SLEIGH processor specification compiler, decoder and lifter), a JIT-based execution engine and an efficient software memory-management unit (MMU) implementation. The SLEIGH runtime handles loading the appropriate SLEIGH specification for the guest architecture, assigning a mapping from guest registers to virtual P-code registers, and then decoding and translating ISA-specific machine code to P-code. Unlike Ghidra's SLEIGH runtime, Icicle's runtime assigns sequential IDs to virtual registers, allowing them to be managed in a dense array, improving performance. We also implement a lightweight P-code optimization pass that performs constant evaluation and dead-code elimination, significantly reducing the number of P-code operations when values are known at translation time. Icicle's JIT-based execution engine, then groups P-code operations in blocks and compiles them to native code using Cranelift [1], an open-source low-level code generation framework. Cranelift provides register allocation, instruction legalization, and additional optimizations. Later, during a recompilation step, multiple blocks are compiled as part of a single compilation unit, enabling additional optimizations that improve performance. Notably, *unlike* existing emulators, Icicle does not discard the P-code representation of each block after JIT compilation. This can significantly aid any analysis used for complex instrumentation, at the cost of some additional memory overhead.

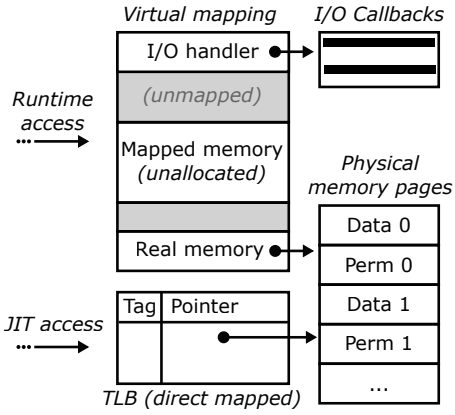

Figure 2: Overview of the byte-level software memory-management unit (MMU) implemented in Icicle.

When the memory layout of the guest is incompatible with the host, it is necessary for the emulator to handle the differences. Therefore, Icicle uses a software MMU to handle guest memory accesses (Figure 2). The software MMU maintains virtual mapping table that allows guest memory and memory mapped IO (MMIO) to be mapped in the emulator. The mapping table is represented as a range-map, allowing for *byte*-level precision at the cost of more expensive lookups. To improve efficiency, we cache translated addresses in a lookup table referred to as a translation lookaside buffer (TLB), named after its hardware analog. The JIT compiled

[3]Ghidra contains a limited P-code emulator and has been used for micro-fuzzing [20] but is unable to satisfy the needs of a full modern fuzzing framework, for example, Ghidra's P-code emulator is interpreter-based hindering performance. Notably, Icicle only uses the SLEIGH specifications and *none* of the components of Ghidra's emulator.

code can directly access guest memory using the TLB allowing for fast execution in most cases. Whenever, an address not in the TLB, is encountered, the JIT calls a runtime helper that handles the access and caches the translated address. To retain byte-level mapping, ICICLE maintains a permission byte for each physical byte which is checked by the JIT on access, similar to approaches used in prior work [22, 26, 58]. Both Unicorn and QEMU (when running in full-system mode) also implement a software MMU. However, they both require memory to be mapped in page-sized (4 KB) regions.

The added flexibility of the byte-level mapping in ICICLE *allows more accurate emulation of embedded system memory and can be used to enable better bug detection.*

In addition to CPU emulation, most binaries interact with external resources such as file systems, hardware, and other software. ICICLE is designed to be flexible and extensible using pluggable *environment emulators* (see Figure 1). To demonstrate the functionality of the system, we have implemented environments to allow comparisons against existing emulation-based fuzzers. The current implementation supports fuzzing Linux userspace binaries by emulating a subset of system calls, supports fuzzing several MSP430 MCUs ISAs, and supports embedded ARM binaries using FUZZWARE [54].

3.2 Architecture-Agnostic Instrumentation

To implement arbitrarily complex fuzzing instrumentation requires: i) the ability to analyse the semantics of a target program; ii) an efficient mechanism to capture runtime information about the running program; iii) a way of sharing the captured information with the fuzzing frontend. Additionally, to be effective in a fuzzing context, these requirements must be supported in a manner that has low performance overheads.

ICICLE supports these requirements through a set of APIs added to the emulator, we refer to instrumentation utilizing these APIs as *Tracer Plugins*. These APIs enable:

- *Direct access to the architecture-agnostic P-code representation of the program.* Plugins are able register a callback function to be called whenever the emulator translates a new block to P-code. The callback function is provided with the full code-cache including the newly translated block, satisfying the first requirement enabling architecture agnostic code analysis.
- *Inline code-injection.* Plugins can inject additional P-code operations into any block enabling inline instrumentation to be supported in an architecture agnostic manner. Any modified blocks are invalided by ICICLE and re-compiled by the JIT the next time they are executed.
- *Registry of JIT and fuzzer accessible shared memory.* During initialization, plugins can register storage locations with the emulator, which can later be manipulated with P-code operations. Additionally, ICICLE allows plugins to define custom P-code registers, these custom registers are treated the same as guest registers for the purpose of register allocation during JIT code-generation, which can allow for more efficient instrumentation in some cases. This enables data to be efficiently saved by injected instrumentation and analyzed as part of the fuzzing loop.

To illustrate expressiveness ICICLE's instrumentation method and its ability to support architecture-agnostic instrumentation, we discuss the implementation of the four techniques discussed in Section. 2.1 in ICICLE and compare them to implementations in other emulation-based fuzzers.

Branch hit counts. In ICICLE, branch hit counts are implemented by a Tracer Plugin that does the following: during initialization, it registers the location of coverage bitmap with the emulator and defines a custom register to store the previous program location. When a new block is translated, the plugin injects code at the start of the block that computes a hash of (current_location, previous_location), which is then used as an index for updating the coverage bitmap. Since the instrumentation is implemented using P-code injections, the JIT can generate native code that updates the coverage bitmap without resorting to a function call. In addition to branch hit counts, ICICLE also implements both block-only coverage and edge coverage.

Existing emulators are also able to add branch hit count instrumentation in an architecture independent manner by injecting code when new translation blocks are created, which is common across architectures in QEMU. However, AFL++'s implementation in both QEMU and Unicorn makes direct modifications the emulator (although the changes are relatively minor). Notably, in AFL++'s Unicorn mode, branch coverage instrumentation is *not* implemented using Unicorn's hooking API, since the instrumentation is highly performance sensitive, and the hooking API imposes additional overheads.

Context-sensitive branch coverage. ICICLE implements context-sensitive branch coverage with a Tracer Plugin. This plugin defines a custom register to store the context, then when a new block ending with a `CALL` is translated, the generates a random value to use as context for the current location and XORs it the context register. In the block after the call, the instrumentation is injected to clear the added context. The branch hit count plugin is then modified to use the context value by using it as part of computing the index into the coverage map. Since the `CALL` hint is part of P-code representation, it allows us to write a portable implementation that works across architectures.

Context-sensitive coverage was first implemented in Angora[12] using compiler-based instrumentation, and in afl-sensitive [64] for binary-only instrumentation using QEMU. afl-sensitive's implementation modifies QEMU's x86 translation layer to add instrumentation that updates the calling context on `call` and `ret` instructions. Since afl-sensitive instruments x86 specific instructions it is not portable to other architectures.

CmpLog. There are two parts to CmpLog, first relevant comparison operations must be identified, and second, the operands of each comparison must be copied to a fuzzer accessible location.

Inspired by the success of Datalog for program analysis tasks [28, 59], we implement a comparison finding algorithm as a set of Datalog rules in Listing 2. Since the rules are defined in terms of P-code operations, it allows ICICLE support CmpLog for any architecture.

In contrast, existing CmpLog implementations require identifying architecture specific instructions to identify comparisons. For example, on x86, AFL++'s instruments `CMP` and `SUB` instructions, by modifying QEMU's translation stage. This has two main issues: 1)

```
% x is an copy of the destination of an operation.
copy(x, x) :- op(x, _, _, _).
% b = a if it is the destination of a copy-like op with a.
copy(a, b) :- op(b, "COPY", a).
copy(a, b) :- op(b, "ZXT", a).
% b = a if x = a and b = x
copy(a, b) :- copy(a, x), copy(x, b).
% Identify p-code operations corresponding to comparisons.
cmp("==", cond, a, b) :- op(cond, "==", a, b).
cmp("!=", cond, a, b) :- op(cond, "!=", a, b).
% `(a - b) [cmp] 0` => `a [cmp] b` (subtract and compare with zero)
cmp(op, cond, a, b) :- op(cond, "-", a, b), cmp(op, cond, x, 0).
% `!(a [inv(op)] b)` => `a [op] b` (inverted comparison)
cmp("!=", cond, x, y) :- op(notc, "!", cond), cmp("==", notc, x, y).
cmp("==", cond, x, y) :- op(notc, "!", cond), cmp("!=", notc, x, y).
% Output comparisons that flow into the branch condition.
output(op, a, b) :- cmp(op, cond, a, b), copy(cond, x), branch(x).
```

Listing 2: Datalog rules for finding comparison operands. The list of p-code operations to analyse, and the branch exit condition are provided as inputs.

since the instrumentation looks for specific instructions, a separate implementation is required for each architecture, 2) it can result in excessive instrumentation, for example most SUB operations on x86 are not used for comparisons. CmpLog is not supported in Unicorn.

CompareCov. In Icicle, integer comparisons are identified using the same algorithm as CmpLog. Once identified, Icicle injects code that writes to the coverage bitmap for each matching byte before the original comparison operation. For memory comparisons functions, Icicle searches for the target functions in the program's symbol table and injects instrumentation when a block calling the target function is translated. This allows Icicle's instrumentation to be used for statically linked binaries including firmware (as long as the symbol table has not been stripped).

In contrast, AFL++'s implementation for integer comparisons requires identifying architecture specific comparison instructions, like CmpLog instrumentation. Further, for memory comparison functions, it relies on the dynamic linker to replace the original comparison functions with instrumented versions. This approach is unable to support instrumenting statically linked firmware binaries.

Summary. Icicle's design enables it to satisfy all fuzzing instrumentation requirements in a manner that is simultaneously: i) efficient; ii) avoids new instrumentation requiring direct modification of the emulator internals; and iii) is architecture agnostic. Each instrumentation technique is implemented targeting P-code enabling it to support any ISA. And, as an added benefit, *only knowledge of P-code is adequate* for developing new instrumentation techniques.

3.3 Fuzzing Frontend Integration

Modern grey-box fuzzing frameworks consists of two main components: the *frontend* which handles input generation, input scheduling, hang detection and crash deduplication, and the *backend* which manages program execution, crash monitoring, and instrumentation. Emulation-based fuzzers utilize emulators as the fuzzing backend allowing for *binary-only* and *cross-architecture* fuzzing. Icicle is a new fuzzing backend, therefore, we make our emulator compatible with an existing fuzzing framework: AFL++ [27] to avoid implementing a new frontend. AFL++ is a state-of-the-art fuzzing framework derived from the well-known American Fuzzy Lop (AFL) [66] project, with general improvements, and support for additional fuzzing techniques. Icicle integrates with AFL++ using the forkserver interface also used by AFL++'s QEMU-mode.

4 EVALUATION

Settings. Unless otherwise specified, all experiments were carried out with AFL++ 4.01a as the fuzzing frontend on an AMD Ryzen Threadripper 3990X restricted to a single core. All AFL++ settings were kept as default, except to enable instrumentation as needed and to adjust the timeout for hang detection.

Experiments. We design our experimental regime to answer five specific questions articulated in Section 4.1-4.5.

4.1 Is Icicle's Instrumentation Portable Across Diverse ISAs?

To ensure that architecture-agnostic instrumentation implemented in our emulator is operational across a range of architectures, we designed a test program, shown in Listing 3, that consists of 5 synthetic bugs designed to test specific instrumentation.

```
void test_instrumentation(char* buf) {
    // (1) comparison against a single byte in the input
    if (buf[0] == '%') {
        crash(1);
    }
    // (2) Multiple comparison against single bytes of the input.
    if (buf[0] == 'i' && buf[1] == 'x' &&
        buf[2] == 'S' && buf[3] == 'D') {
        crash(2);
    }
    // (3) A single comparison against multiple input bytes.
    if (*(u32*)buf == *(u32*)"wzfc") {
        crash(3);
    }
    // (4) A multi-byte comparison across a function call.
    if (compare(buf, "dGlIHF1W") == 0) {
        crash(4);
    }
    // (5) Saturate coverage then compare.
    saturate_compare2_cov();
    u32 tmp = *((u32*)buf) ^ 0x46092d5f;
    if (compare2(tmp, 0x7451496b)) {
        crash(5);
    }
}
```

Listing 3: Test program used for evaluating instrumentation.

We evaluate the portability of Icicle's instrumentation by fuzzing the test program compiled for 5 different architectures. For architectures with Linux support, we configure the program to read the input from `stdin`. For MSP430, the program reads from a peripheral mapped to the fuzzing input. After compiling the binary for each architecture, we manually verified that the machine code of output binary behaves as expected. As a baseline we compare against AFL++'s QEMU-mode when instrumentation is supported for the guest architecture. For each fuzzing configuration, we perform 20 trials for a maximum of 10 minutes starting with an uninformed seed. The results from this experiment are shown in Table 1.

Both *Bug1* and *Bug2* are discoverable with code coverage alone, so are found by all fuzzing configurations. *Bug3* requires additional

Table 1: Results from different fuzzing instrumentation configurations for the test program. ✓ denotes the bug ID was found at least once within 10 minutes. Each test was repeated 20 times. Shaded grey areas are due to: i) unsupported fuzzing instrumentation for MIPS and RISC-V in QEMU emulation with AFL++; and ii) MSP430 ISA being unsupported in QEMU.

		x86-64					AArch64					MIPS					RISC-V					MSP430				
Fuzzer	Instrumentation	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Icicle (ours)	Cov	✓	✓	-	-	-	✓	✓	-	-	-	✓	✓	-	-	-	✓	✓	-	-	-	✓	✓	✓	-	-
	Cov+CmpLog	✓	✓	✓	✓	-	✓	✓	✓	✓	-	✓	✓	✓	✓	-	✓	✓	✓	✓	-	✓	✓	✓	✓	-
	Cov+CompareCov	✓	✓	✓	-	-	✓	✓	✓	-	-	✓	✓	✓	-	-	✓	✓	✓	-	-	✓	✓	✓	-	-
	Cov+Context	✓	✓	-	-	✓	✓	✓	-	-	✓	✓	✓	-	-	✓	✓	✓	-	-	✓	✓	✓	✓	-	✓
QEMU	Cov	✓	✓	-	-	-	✓	✓	-	-	-	✓	✓	-	-	-	✓	✓	-	-	-					
	Cov+CmpLog	✓	✓	✓	✓	-	✓	✓	✓	-	-															
	Cov+CompareCov	✓	✓	✓	-	-	✓	✓	✓	-	-															

instrumentation to be found so can only be found when one of the two comparison instrumentation techniques is enabled, except for the MSP430 binary. With CmpLog, the fuzzer can find a solution via an input-to-state mutation directly replacing the incorrect value with a correct one, with CompareCov enabled, the comparison is split into byte-level comparisons, and the fuzzer observes incremental coverage feedback similar to *Bug2*. Since MSP430 is a 16-bit architecture, the compiler splits the 32-bit comparison into two 16-bit comparisons allowing the fuzzer to eventually find the crashing input for *Bug3* without additional instrumentation. *Bug4* evaluates the fuzzers ability to solve memory comparison functions so is only discovered when CmpLog instrumentation enabled, which generally finds the crashing input within seconds. CompareCov fails to find the bug, since `compare` is not a standard comparison function and is therefore not instrumented by CompareCov. CmpLog is only partially implemented for QEMU on AArch64 (function calls are not instrumented) so fails to find *Bug4*. *Bug5* tests the fuzzer's ability to find a bug in a function where code coverage is saturated by a previous call, so is only found when context sensitive branch coverage is enabled, which only Icicle supports on all architectures.

Summary The test program binaries for five different ISAs provide empirical evidence that the architecture agnostic instrumentation implementation of the different instrumentation techniques in Icicle is both effective and portable across architectures.

4.2 Is Architecture-Agnostic Instrumentation as Effective as Existing Architecture-Specific Implementations?

LAVA-M [21] is a widely used set of binaries for evaluating and benchmarking fuzzers. It consists of four binaries from GNU coreutils [29] each injected with synthetic bugs. While the injected bugs are not representative of typical real-world vulnerabilities [37], previous work has demonstrated that these bugs are difficult to find with code coverage only but can be found with by instrumenting comparison operations [5, 12, 30]. This naturally lends itself to assessing Icicle's architecture-agnostic implementation of CmpLog and CompareCov instrumentation.

Using AFL++ as the frontend, we evaluate the bug discovery capability of Icicle across four different ISAs (x86, AArch64, MIPS, and RISC-V), the first two of which we compare against QEMU[4]. We also evaluate both QEMU and Icicle on x86 with code coverage only as a baseline. For each of the injected bugs, a unique ID is written to `stdout` whenever the bug is triggered. Therefore, we can verify each crash by running the x86 version of the binary on the host machine then checking for unique bug IDs in `stdout`. We perform 5 trials for each fuzzing configuration running for 12 hours each, starting with the same two initial seeds as Section 4.1. Figure 3 shows the bugs found over-time with instrumentation enabled and Table 4 in the Appendix details the total bugs found for every architecture, instrumentation, and emulator configuration.

With code coverage alone almost no bugs are found by either emulator in any of the binaries. Both comparison instrumentation techniques allow most bugs to be found, with CmpLog finding bugs significantly faster than CompareCov in several cases. Icicle's results closely match QEMU results for both AArch64 and x86, which supports our claim that Icicle's instrumentation is as effective as the architecture-specific approach employed by AFL++'s QEMU-mode. On the two additional architectures tested with Icicle both instrumentation techniques continue to be effective. However, the results for the MIPS version of `uniq` are slightly worse, this is caused by differences in the memory layout (MIPS uses a 32-bit address space, while the other architectures are 64-bit), which causes issues when replaying the crashing input on the x86 host.

The differences in the number of crashes found for `who` binary across architecture is caused caused by platform specific behaviour in the program itself. The fuzz input is parsed as a `utmpx` structure, however the layout of the fields within the structure is different across architectures. This can cause certain bugs to become unreachable, and causes issues when we attempt to replay the crashing inputs on the x86 version of the binary in order to verify the crash IDs. Additionally, the binary frequently crashes before a bug ID is flushed to *stdout* (caused by internal buffering), which prevents us from obtaining the bug ID from the original execution. Notably, all bugs reported and discovered are those reproduced on both the guest architecture and the host (x86). This is additional evidence of the importance of *binary-only* and *cross-architecture* fuzzing; even when source code is available, program behaviour can differ on between architectures.

[4]We compare with QEMU not Unicorn, since Unicorn cannot directly execute Linux binaries. Additionally, since neither CmpLog nor CompareCov instrumentation are supported in AFL++'s QEMU-mode for RISC-V and MIPS, we only evaluate these architectures with Icicle

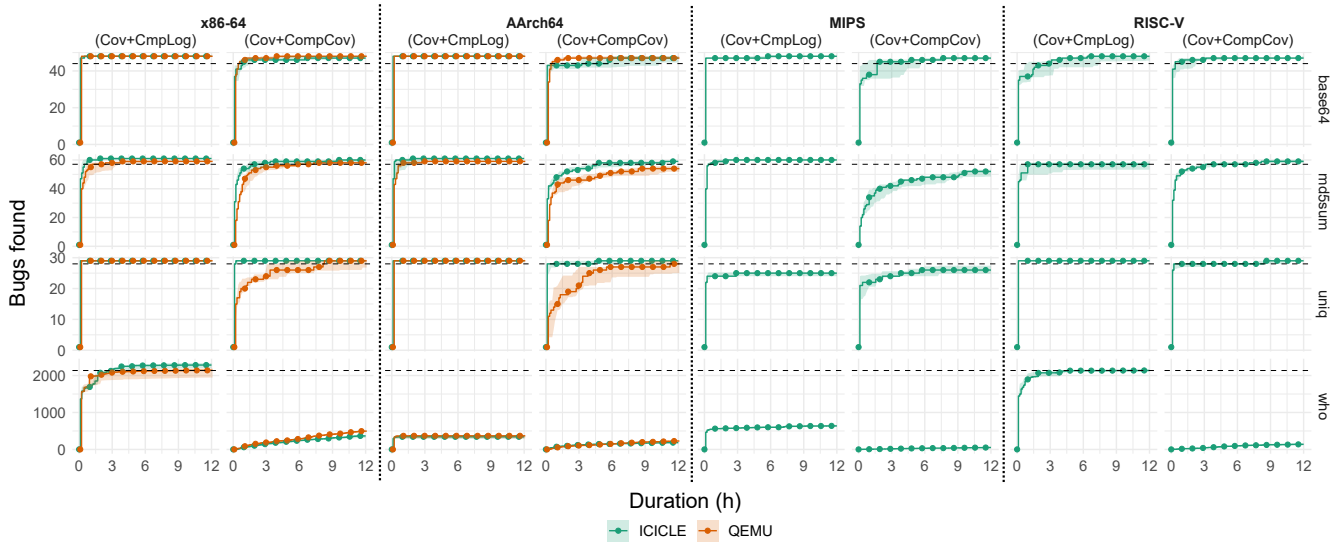

Figure 3: LAVA-M bugs found over time in each binary. The solid line represents the median number of bugs found, the shaded area represents the min/max coverage across all trials, and the black dotted lines represent the number of bugs listed in the LAVA-M paper (*Note*: it is well known that it is possible to trigger additional bugs other than specified in the original paper).

Summary Discovering LAVA-M benchmark bugs require a specific operational capability from instrumentation techniques to solve comparison operations; namely CompCov or CompLog. Icicle's results closely match QEMU results for both AArch64 and x86, supporting our claim that Icicle's instrumentation is as effective as the architecture-specific approach employed by AFL++'s QEMU-mode. On MIPS and RISC-V architectures (where AFL++'s QEMU-mode does not support the necessary instrumentation) both instrumentation techniques tested with Icicle continue to be effective.

4.3 Can Icicle Be Used to Implement and Enhance State-of-the-Art Firmware Fuzzing Techniques?

Fuzzware [54] is a recent state-of-the-art fuzzing framework for analyzing ARM firmware binaries. Fuzzware extends Unicorn to instrument and execute ARM firmware. We replace Unicorn with Icicle to evaluate Icicle's ability to support state-of-the-art firmware fuzzing. We then tested our modified version (Fuzzware-Icicle) by attempting to reproduce Fuzzware's results on the 10 *real-world* binaries used in the P^{2}IM [24] firmware set as they were evaluated extensively by [24, 69] and Fuzzware. Importantly, since Icicle's instrumentation is portable we can support *additional instrumentation* when fuzzing ARM firmware. In particular, we perform additional tests with CompareCov instrumentation enabled to allow for better comparison solving[5]. We followed the same experimental setup for Fuzzware as described in the original paper (we used the same number of trials, seeds and run time duration).

We were able to successfully rediscover all 16 of the bugs found by Fuzzware, and, with CompareCov enabled, Fuzzware-Icicle was able to find an additional bug in the Console binary *not reported* by any prior work[6]. As part of the `rtc settime` command, the firmware reads a date from the user in the form `YYYY-MM-DD HH:MM:SS` without checking whether the parsed date is valid. This causes an out-of-bounds access when the name of the month is resolved using a lookup table. Since reaching this bug requires first solving a string comparison to reach the `rtc` handler, then solving a second string comparison for the `settime` subcommand, we believe the added instrumentation was critical to finding this bug.

Icicle also found an additional crash in the Soldering Iron binary. At high temperatures, rendering the heat indicator causes the buffer allocated for the LCD screen to overflow. However, after further analysis we discovered the maximum temperature is *restricted* in software, indicating that the bug is a false-positive caused by Fuzzware's peripheral modelling strategy.

In addition to reproducing the bugs, we also investigated whether Fuzzware-Icicle is able to maintain the same level of block coverage as the original implementation. The results are shown in Figure 4. For almost all the evaluated binaries we achieve almost identical block coverage to Fuzzware, with some small differences of which we manually investigated. With CompareCov enabled, Fuzzware-Icicle achieves higher coverage in two of the binaries: Console, and Steering Control. The higher coverage in Console corresponds to reaching different command handlers that are dispatched based on string comparisons, including the sub-commands of the `rtc` handler that contains the bug discussed above. Similarly, Steering Control contains two commands, that are triggered when the matching string is read by the firmware (`"steer"`, and

[5]We did not test with CmpLog, since effective use of the instrumentation requires additional integration with fuzzing frontend, unsupported by Fuzzware.

[6]Crashing inputs for each of the discovered bugs are available in our GitHub repository.

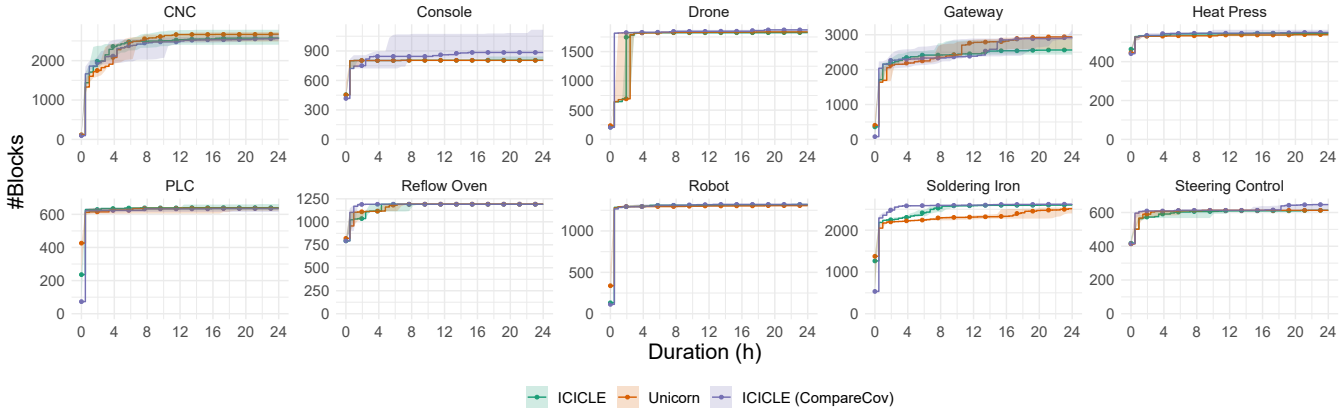

Figure 4: Block coverage over time for the *real-world* ARM firmware using the two different emulators and CompareCov instrumentation supported in ICICLE. The solid line represents the median coverage of 5 runs, and the shaded area represents min/max coverage.

`"motor"`). CompareCov enables ICICLE to generate inputs containing the command strings, and thus is able to reach additional code. The discrepancies in the Gateway and Soldering Iron binaries are caused by high variance between fuzzing runs, running additional trials would likely remove any discrepancies.

Summary ICICLE is a robust emulator capable of supporting the current state-of-the-art ARM firmware fuzzer, FUZZWARE. We discovered all 16 known bugs. ICICLE provides a direct substitute for Unicorn with the added advantage of additional, architecture agnostic instrumentation shown to be effective at *improving coverage* and *discovering 2 new bugs* in real-world firmware not reported by fuzzing efforts in prior work.

4.4 Can ICICLE Discover Bugs in Binaries in an ISA Currently Not Supported by Emulation-Based Fuzzers?

To demonstrate the architecture-independent benefits of our prototype emulator, we investigate fuzzing firmware written for MSP430 microcontrollers. FiE [18], is the only prior study that attempted to find bugs in MSP430 firmware. However, FiE requires C source-code and therefore does not support manually written assembly code (which is common in larger firmware) and is incapable *binary-only* fuzzing. Further, MSP430 firmware is not supported by any existing emulation-based fuzzing framework[7], and therefore presents a compelling use case for fuzzing with ICICLE.

Similar to existing monolithic firmware fuzzing approaches [24, 54], we handle peripheral accesses for MSP430 firmware by reading them from the fuzzer input. We found this highly effective at finding bugs. We selected 3 different firmware to evaluate. First, inspired by FiE, we evaluated the USB SDK provided as part of TI's MSP430 USB Developers Package[8] using example programs provided as part of the development package (H4_PacketProtocol) as a harness. Second, we compiled an unmodified version of the Goodwatch [63] firmware[9], a hardware and firmware replacement for Casio calculator watches based on the CC430 MCU (a MSP430 CPU with an integrated RF transceiver) as that was the highest-ranking application on GitHub's trending page for MSP430. Additionally, we investigated ICICLE's ability to test closed-source firmware by extracting the firmware off a commercial medical device, a Polar heart rate tracker, containing a MSP430F2132 MCU as shown in Figure 5.

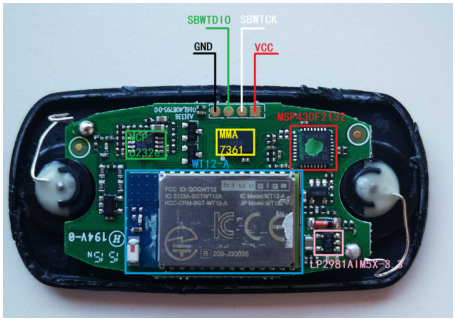

Figure 5: The internals of a dismantled Polar Heart Rate Tracker.

Table 2: Discovered bugs in MSP430 *real-world* binaries.

Firmware	Bug description (PoCs & stack traces on GitHub)
Goodwatch	Incorrect comparison when writing to log buffer.
Goodwatch	Buffer overflow when handling zero length packet.
Goodwatch	Stack overflow in RNG generation.
Goodwatch	Out-of-bounds access in OOK keypress.
Goodwatch	Out-of-bounds access in Stopwatch.
H4_PacketProtocol	Unchecked Interface Index in Get Descriptor.
H4_PacketProtocol	Buffer overflow in Set Report.

[7] [38] is a fork of QEMU adding MSP430 support, however it is outdated, not integrated with any fuzzing framework, and does not support MSP430 CPUX extensions.
[8] https://www.ti.com/tool/MSP430USBDEVPACK version 5.20.06.03
[9] commit: c8859f845fccf56585a127059b1d1b825b381673

Table 3: Block coverage (#BB) for the real-world MSP430 binaries with and without CmpLog instrumentation enabled. Avg represents the median coverage achieved after 24 hours in 5 trials.

Firmware	#BB total	Instrumentation	#BB min	#BB avg	#BB max
Goodwatch	3263	Cov	2336	2362	2441
		Cov+CmpLog	**2438**	**2503**	**2526**
H4 Packet Protocol	925	Cov	**819**	821	891
		Cov+CmpLog	813	**910**	**914**
Heart Rate Tracker	744	Cov	679	679	716
		Cov+CmpLog	**680**	**717**	**718**

The block coverage results are summarized Table 3. After triaging the results, we identified two unique bugs H4_PacketProtocol and 5 unique bugs in the Goodwatch firmware and 3 additional crashes related to debugging features. While no bugs were discovered for the Polar heart rate tracker, the fuzzer reached almost all blocks in the firmware. The bugs discovered by Icicle are summarized in Table 2, and for each bug discovered we provide input files and a detailed crash analysis in our GitHub repository[10].

Summary MSP430 firmware fuzzing is not supported by existing emulation-based fuzzing frameworks. Case studies with Icicle and its suite of architecture agnostic instrumentation discovered seven undiscovered software bugs in two (USB SDK–H4_PacketProtocol, and Goodwatch) of the three tested MSP430 binaries.

4.5 How Does Icicle Perform in Fuzz Testing?

In the development of Icicle, we made efforts to ensure that Icicle has good performance in general. We compared *fuzz test execution speed* of Icicle with Unicorn (emulator) employed by the sate-of-the-art fuzzer, Fuzzware, on the P^{2}IM dataset [24] and summarise the results in Figure 6.

Summary Icicle has approximately the same performance as Unicorn for fuzzing monolithic firmware binaries.

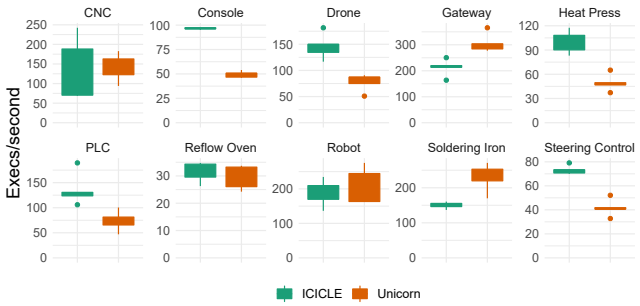

Figure 6: Icicle and Unicorn performance comparison when integrated with the state-of-the-art fuzzer, Fuzzware.

[10] https://github.com/icicle-emu/icicle/tree/main/crash-analysis/msp430

5 DISCUSSION AND LIMITATIONS

Although we have taken the first steps to re-think and re-design an emulation framework to directly support fuzzing requirements, and instrumentation development and testing, the current implementation is not without limitations. The released emulator prototype was primarily designed for CPU ISA emulation, similar to the goals of the Unicorn project. As a result, Linux emulation is minimal, and more complex hardware emulation required for full-system emulation (e.g., page-table emulation) is not currently supported.

5.1 Emulator Correctness

In emulation-based fuzzing, since the program not executed on the original hardware, there is a risk that any crashes discovered could be caused by emulation issues, not bugs in the target program. To reduce the chance of emulation bugs in Icicle, first, we employ a *differential testing* strategy, similar to other widely used approaches for testing CPU emulators [2, 34, 43–45, 65]. Second, we manually investigated any crashes discovered in benchmark evaluations and ensure they are caused by program bugs.

5.2 Performance

In the development of Icicle, while we made efforts to ensure that Icicle has good performance in general, there are several additional optimizations possible. The current implementation of Icicle has demonstrably similar performance to Unicorn (see Figure 6). Although a direct performance comparison against QEMU is desirable, it is more difficult. Icicle implements a forkserver similar to AFL++'s persistent mode, however we run AFL++'s QEMU-mode without this feature since (currently) persistent mode requires a significant amount of manual effort to set up for each binary (notably, Icicle's implementation is automatic for Linux binaries). This results in Icicle performing significantly faster for small binaries. Other the other hand, since Icicle always translates memory accesses in software (like Unicorn) while AFL++'s QEMU-mode can utilize hardware address translation when running Linux user-space binaries on a Linux host, we expect a speedup for QEMU for larger Linux binaries.

6 RELATED WORK

Improving emulation-based fuzzing. There has been some effort in improving QEMU and Unicorn for fuzzing, including, improving runtime performance [6, 27], enabling support for full-system emulation of Linux-based firmware [11, 16, 61, 67, 68], and extending the emulator to support additional analysis such as taint tracking and symbolic execution [53, 60].

Binary-only fuzzing. Without access to source-code it is challenging to use fuzzing techniques that rely on instrumentation, since the simplest approach using compiler-based code injection, is not possible. Fuzzers that support targets without source-code are known *binary-only* fuzzers. Emulation-based approaches are one solution, however there are several other alternatives.

Virtualisation/hardware-assisted approaches (e.g., kAFL [56], and NyX [55, 57]) use a variety of hardware features to implement fuzzing instrumentation. Since they require additional hardware support some instrumentation cannot be easily implemented,

and firmware fuzzing is not supported. Static rewriting techniques (e.g., Retrowrite [19], Datalog Disassembly [28], Zafl [48]) disassemble a binary, inject instrumentation, then reassemble the binary. This can enable close to compiler-level instrumentation performance, however the complexity involved in the rewriting process often results in correctness issues, typically firmware binaries are not well supported, and static rewriting cannot be used cross-architecture fuzzing. While, dynamic instrumentation tools (e.g., DynamoRIO [9], PIN [39], CMU BAP [10], Valgrind [49]), share significant similarities to emulation-based approaches, they are more restrictive than full emulators and are unable to support firmware fuzzing.

Embedded system fuzzing. Fuzzing embedded systems and IoT devices is difficult because we cannot avoid dealing with hardware and peripheral interactions since it might represent a majority of the code we are trying to test. As a result, emulation-based fuzzers need to support more than just CPU emulation. Past work has extended either QEMU or Unicorn to support firmware fuzzing through, hardware-in-the-loop approaches [17], peripheral modeling [18, 24, 32, 69], or emulating the hardware abstraction layer [15]. More recently, Fuzzware [54] investigated techniques to automatically generate peripherals models using local symbolic execution. More recently, Ember-IO [23] removed the need for peripherals models and exploited commonalities in the way hardware behaves to make better use of input data while minimizing the impact of poor coverage feedback resulting from excess coverage caused by interrupts. Both Fuzzware and Ember-IO demonstrated results outperforming prior firmware fuzzing approaches. Notably, aside from FiE (which is source-based and only targets MSP430), all these approaches only evaluate firmware written for the ARM architecture and either employ Unicorn or QEMU. Icicle makes it easier to fuzz multiple architectures, which we hope will assist in increasing the architecture diversity in future work. MetaEmu [13] also uses SLEIGH to support emulating multiple architectures, however, MetaEmu's primary goal is to support general dynamic binary analysis, not high-performance fuzzing, and as such lacks a JIT-based execution model, instead focusing on supporting flexible execution modes.

7 CONCLUSION

Emulation-based fuzzing techniques, supported by effective instrumentation, are highly flexible and are the only method for *cross-architecture* fuzzing. For historical reasons, emulators used in existing emulation-based fuzzing frameworks were not designed for fuzzing and has made it difficult to meet fuzzing specific needs such as implementing advanced instrumentation techniques supporting a design-build-and-test once-only paradigm across multiple ISAs and implementing fuzzing specific optimizations.

We designed and implemented a new multi-architecture emulation framework for fuzzing. Within our framework, we implemented four different architecture agnostic instrumentation techniques and demonstrated that a single architecture-independent implementation is effective across multiple architectures. Our emulation platform is extremely flexible, supporting a wide range of ISAs, especially significant in fuzzing firmware in embedded systems and IoT devices. This was demonstrated by discovering 7 new bugs in ARM firmware by integrating with the state-of-the-art ARM firmware fuzzer, Fuzzware and fuzzing firmware for MSP430 ISAs—an unsupported target by existing emulation-based fuzzers.

8 DATA AVAILABILITY STATEMENT

We uploaded artifacts to https://github.com/icicle-emu/icicle, including source code, a README guide for users, proof-of-crash (PoC) inputs and stack traces for all discovered bugs.

ACKNOWLEDGEMENTS

The work has been supported by the Australian Government's Research Training Program Scholarship (RTPS) and Cyber Security Research Centre Limited whose activities are partially funded by the Australian Government's Cooperative Research Centres Programme.

APPENDIX

Table 4: Total LAVA-M bugs found within 5 trials of each configuration over 12 hours. The results demonstrate the effectiveness of the architecture agnostic instrumentation as well as the benefits from making available state-of-the-art instrumentation across architectures with ease.

ISA	Fuzzer	Instrumentation	LAVA-M bugs found			
			base64	md5sum	uniq	who
x86-64	ICICLE	Cov	4	0	1	0
		Cov+CmpLog	48	61	29	2419
		Cov+CompareCov	48	60	29	1002
	QEMU	Cov	0	0	1	0
		Cov+CmpLog	48	59	29	2356
		Cov+CompareCov	48	59	29	1256
AArch64	ICICLE	Cov	0	0	2	0
		Cov+CmpLog	48	61	29	375
		Cov+CompareCov	48	60	29	357
	QEMU	Cov	2	0	0	0
		Cov+CmpLog	48	59	29	389
		Cov+CompareCov	48	59	29	381
MIPS	ICICLE	Cov	1	0	1	0
		Cov+CmpLog	48	61	28	799
		Cov+CompareCov	48	60	29	148
	QEMU	Cov	0	0	0	0
RISC-V	ICICLE	Cov	1	0	0	0
		Cov+CmpLog	48	59	29	2327
		Cov+CompareCov	48	60	29	472
	QEMU	Cov	1	0	0	0

REFERENCES

[1] Bytecode Alliance. 2021. Cranelift Code Generator. https://github.com/bytecodealliance/wasmtime/tree/main/cranelift.

[2] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. 2015. Virtual CPU Validation. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. 311–327. https://doi.org/10.1145/2815400.2815420

[3] Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: High-Throughput Fuzzing of Network Applications. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*. 340–351. https://doi.org/10.1145/3533767.3534376

[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Network and Distributed Systems Security Symposium (NDSS '19)*. https://doi.org/10.14722/ndss.2019.23412

[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS '19)*. 1–15. https://doi.org/10.14722/ndss.2019.23371

[6] Andrea Biondo. 2018. Improving AFL's QEMU mode performance. https://abiondo.me/2018/09/21/improving-afl-qemu-mode.

[7] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security '19)*. 1985–2002. https://doi.org/10.5555/3361338.3361475

[8] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *44th IEEE/ACM International Conference on Software Engineering (ICSE '22)*. https://doi.org/10.1145/3510003.3510230

[9] Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE '23)*. 133–144. https://doi.org/10.1145/2151024.2151043

[10] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*. 463–469. https://doi.org/10.5555/2032305.2032342

[11] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards automated dynamic analysis for linux-based embedded firmware. In *Network and Distributed System Security Symposium (NDSS)*. https://doi.org/10.14722/ndss.2016.23415

[12] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (SP)*. 711–725. https://doi.org/10.1109/SP.2018.00046

[13] Zitai Chen, Sam L Thomas, and Flavio D Garcia. 2022. MetaEmu: An Architecture Agnostic Rehosting Framework for Automotive Firmware. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. 515–529. https://doi.org/10.1145/3548606.3559338

[14] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. 265–278. https://doi.org/10.1145/1950365.1950396

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

[16] The Qiling Contributors. 2022. Qiling Advanced Binary Emulation Framework. https://github.com/qilingframework/qiling/

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

[18] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FiE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *22nd USENIX Security Symposium (USENIX Security '13)*. 463–478. https://doi.org/10.5555/2534766.2534806

[19] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy (SP '20)*. 1497–1511. https://doi.org/10.1109/SP40000.2020.00009

[20] Flavian Dola. 2021. Fuzzing exotic arch with AFL using ghidra emulator. https://airbus-cyber-security.com/fuzzing-exotic-arch-with-afl-using-ghidra-emulator/.

[21] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-scale Automated Vulnerability Addition. In *IEEE Symposium on Security and Privacy (SP '16)*. 110–121. https://doi.org/10.1109/SP.2016.15

[22] Brandon Falk. 2018. Vectorized Emulation: MMU Design. https://gamozolabs.github.io/fuzzing/2018/11/19/vectorized_emulation_mmu.html.

[23] Guy Farrelly, Michael Chesser, and Damith C. Ranasinghe. 2023. Ember-IO: Effective Firmware Fuzzing with Model-Free Memory Mapped IO. In *Proceedings of 18th ACM ASIA Conference on Computer and Communications Security (AsiaCCS '23)*.

[24] Bo Feng, Alejandro Mera, and Long Lu. 2020. P^{2}IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *29th USENIX Security Symposium (USENIX Security '20)*. 1237–1254.

[25] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*. 1–13. https://doi.org/10.1145/3395363.3397372

[26] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. 2020. Fuzzing Binaries for Memory Safety Errors with QASan. In *IEEE Secure Development Conference (SecDev '20)*. 23–30. https://doi.org/10.1109/SecDev45635.2020.00019

[27] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT '20)*. https://doi.org/10.5555/3488877.3488887

[28] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog Disassembly. In *29th USENIX Security Symposium (USENIX Security '20)*. 1075–1092. https://doi.org/10.5555/3489212.3489273

[29] Free Software Foundation. 2022. GNU core utilities. https://www.gnu.org/software/coreutils/.

[30] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2577–2594.

[31] 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. https://doi.org/10.1109/ACCESS.2020.2973043

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

[33] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020). https://doi.org/10.1145/3428334

[34] Vladimir Herdt, Daniel Große, Hoang M Le, and Rolf Drechsler. 2019. Verifying Instruction Set Simulators using Coverage-guided Fuzzing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE' 19)*. 360–365. https://doi.org/10.23919/DATE.2019.8714912

[35] Intel. 2016. Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/.

[36] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. 2016. Embedded Security Testing with Peripheral Device Caching and Runtime Program State Approximation. In *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE '16)*.

[37] George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS '18)*. 2123–2138. https://doi.org/10.1145/3243734.3243804

[38] Draper Laboratory. 2016. QEMU MSP430 Target. https://github.com/draperlaboratory/qemu-msp.

[39] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. 190–200. https://doi.org/10.1145/1064978.1065034

[40] Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. 2022. PrIntFuzz: fuzzing Linux drivers via automated virtual device simulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*. 404–416. https://doi.org/10.1145/3533767.3534226

[41] Dominik Maier, Benedikt Radtke, and Bastian Harren. 2019. Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing. In *13th USENIX Workshop on Offensive Technologies (WOOT '19)*. https://doi.org/10.5555/3359043.3359051

[42] Dominik Maier, Lukas Seidel, and Shinjo Park. 2020. BaseSAFE: Baseband SAnitized Fuzzing through Emulation. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '20)*. 122–132. https://doi.org/10.1145/3395351.3399360

[43] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. 337–348. https://doi.org/10.1145/2150976.2151012

[44] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. Testing System Virtual Machines. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*. 171–182. https://doi.org/10.1145/1831708.1831730

[45] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU emulators. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA' 09)*. 261–272. https://doi.org/10.1145/1572272.1572303

[46] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. 1393–1403. https://doi.org/10.1145/3468264.3473932

[47] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar^2: A multi-target orchestration platform. In *Workshop on Binary Analysis Research (BAR '18)*. https://doi.org/10.14722/bar.2018.23017

[48] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. 2021. Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing. In *30th USENIX Security Symposium (USENIX Security '21)*. 1683–1700.

[49] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2007), 89–100. https://doi.org/10.1145/1250734.1250746

[50] Anh Quynh Ngyuen. 2020. Unicorn 2 - looking for sponsors. https://github.com/unicorn-engine/unicorn/issues/1217.

[51] Anh Quynh Ngyuen and Hoang Vu Dang. 2015. Unicorn: Next Generation CPU Emulator Framework. http://www.unicorn-engine.org/BHUSA2015-unicorn.pdf.

[52] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *29th USENIX Security Symposium (USENIX Security '20)*. 2289–2306. https://doi.org/10.5555/3489212.3489341

[53] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *Network and Distributed System Security Symposium (NDSS '22)*.

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

[55] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *30th USENIX Security Symposium (USENIX Security '21)*. 2597–2614.

[56] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security '17)*. 167–182. https://doi.org/10.5555/3241189.3241204

[57] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2022. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys '22)*. 166–180. https://doi.org/10.1145/3492321.3519591

[58] seal9055. 2022. SFUZZ: High Performance Coverage-guided Greybox Fuzzer with Custom JIT Engine. https://seal9055.com/blog/fuzzing/sfuzz.

[59] Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for fast and easy program analysis. In *Datalog Reloaded - First International Workshop (Lecture Notes in Computer Science)*. 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

[60] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security (ICISS '08)*. 1–25. https://doi.org/10.1007/978-3-540-89862-7_1

[61] Jack Tang and Moony Li. 2016. Project Triforce: Run AFL on Everything! *Black Hat Europe* (2016).

[62] The National Security Agency (NSA). 2019. Ghidra: Software Reverse Engineering Framework. https://ghidra-sre.org/.

[63] Goodspeed Travis. 2021. Goodwatch. https://github.com/travisgoodspeed/goodwatch.

[64] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID '19)*. 1–15.

[65] Qiuchen Yan and Stephen McCamant. 2018. Fast PokeEMU: Scaling Generated Instruction Tests Using Aggregation and State Chaining. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '18)*. 71–83. https://doi.org/10.1145/3186411.3186417

[66] Michal Zalewski. 2010. American Fuzzy Lop: a security-oriented fuzzer. https://github.com/google/AFL. https://lcamtuf.coredump.cx/afl/

[67] 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. https://doi.org/10.5555/3361338.3361415

[68] Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. 2022. Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*. 417–428. https://doi.org/10.1145/3533767.3534414

[69] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. In *30th USENIX Security Symposium (USENIX Security '21)*. 2007–2024.

Received 2023-02-16; accepted 2023-05-03