Fuzzing Embedded Systems using Debug Interfaces

Max Eisele
MaxCamillo.Eisele@bosch.com
Robert Bosch GmbH
Renningen, Germany
Saarland University
Saarbrücken, Germany

Daniel Ebert
Daniel.Ebert@bosch.com
Robert Bosch GmbH
Stuttgart, Germany

Christopher Huth
Christopher.Huth@bosch.com
Robert Bosch GmbH
Renningen, Germany

Andreas Zeller
zeller@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

ABSTRACT

Fuzzing embedded systems is hard. Their key components – microcontrollers – are highly diverse and cannot be easily virtualized; their software may not be changed or instrumented. However, we observe that many, if not most, microcontrollers feature a *debug interface* through which a debug probe (typically controllable via GDB, the GNU debugger) can set a limited number of *hardware breakpoints.* Using these, we extract partial coverage feedback even for uninstrumented binary code; and thus enable *effective fuzzing for embedded systems* through a generic, widespread mechanism. In its evaluation on four different microcontroller boards, our prototypical implementation GDBFuzz quickly reaches high code coverage and detects known and new vulnerabilities. As it can be applied to any program and system that GDB can debug, GDBFuzz is one of the least demanding and most versatile coverage-guided fuzzers.

CCS CONCEPTS

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

KEYWORDS

embedded systems, firmware, security, automated software testing, fuzzing, GDB

ACM Reference Format:
Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. 2023. Fuzzing Embedded Systems using Debug Interfaces. 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, 12 pages. https://doi.org/10.1145/3597926.3598115

1 INTRODUCTION

Fuzzing —generating massive amounts of inputs to test a system's robustness—has become the method of choice to detect vulnerabilities in programs. Most modern fuzzers follow the AFL model of fuzzing [36], starting with a population of *seed inputs* which they continuously evolve via small-scale mutations, guided by coverage in the system or software under test. Applying such fuzzers to *embedded systems,* however, is hard. One reason for this is the high diversity in terms of microprocessors, architectures, and operating systems. Most important, however, is that *the software on a microcontroller board is not easily changed,* thus preventing instrumentation. Even when instrumentation is possible, the board needs to provide storage space to capture coverage and other runtime information; and finally, this information needs to find a way back to the fuzzer through some hardware interface [7]. Hence, setting up common fuzzers like AFL on hardware requires implementing *individual coverage collection solutions* for every board.

A large share of recently published embedded fuzzing approaches therefore *virtualize* the embedded system [15, 59]. Such virtualization, however, requires a tradeoff between speed and fidelity [17, 58]. Worse even, it requires not only virtualization of the microprocessor itself, but also of *all other hardware components on the board* as well as virtualizing the way they communicate with each other. Given the enormous *diversity* of available hardware peripherals [26], this requires considerable setup costs, if possible at all [17, 58]. So called *peripheral modeling* approaches [18] try to automate the emulation of peripherals but in our experience *fail* for any interface beyond serial ports.[1]

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

The key idea of this paper is that by systematically *setting breakpoints* in the code based on the control flow graph of the program and by *checking which inputs trigger which breakpoints,* we can

[1]We show this in Section 6.2.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598115

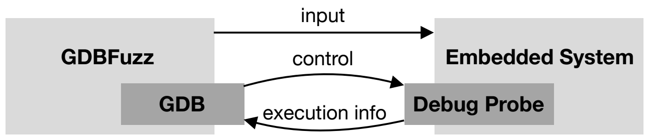

Figure 1: How GDBFuzz works. GDBFuzz leverages a *Debug Probe* that is connected to an *Embedded System* to control execution—notably to set (hardware) breakpoints and detect which inputs trigger which code blocks, and thus obtain coverage without having to instrument or virtualize firmware.

retrieve coverage information, and thus provide the necessary guidance for a feedback-driven fuzzing strategy. As the number of hardware breakpoints within a microcontroller is limited, we set them to a subset of the program's code blocks only, and relocate them periodically. Since many debug probes are addressable via the GNU Debugger (GDB), we have implemented the above strategy in a fuzzer named GDBFuzz, which can leverage GDB interfaces in *any* system to systematically generate test inputs guided by coverage. The required setup is depicted in Figure 1.

Figure 2 summarizes the GDBFuzz operation. Based on the CFG of the target program, GDBFuzz sets the available hardware breakpoints to randomly chosen nodes from the CFG that are yet unreached. GDBFuzz then repeatedly generates input, sends it to the target device, and checks if it triggers a breakpoint indicating new code coverage, or if it crashes the target system.

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

The remainder of this paper is organized as follows. Section 2 discusses the state of the art. Section 3 explains CFG algorithms and CFG extraction of binaries. Section 4 presents the design of GDBFuzz; Section 5 describes implementation details. We evaluate our work in Section 6 and further discuss the results in Section 7. Section 8 closes with conclusion and future work. GDBFuzz is available as open source.

2 BACKGROUND

2.1 Coverage-Guided Fuzzing

Decades ago, Miller *et al.* [37] tested Unix command line tools with random data, observed crashes resulting from software bugs, and called this method "fuzzing". By design, fuzzing can only detect bugs in code that is actually executed, which is why reaching a high code coverage is desired. Nowadays, tons of different fuzzing techniques have been developed, mainly divided into *model-based fuzzing,* where test data is generated based on a specification of the input language, and *mutation-based fuzzing,* where known inputs to the program are randomly mutated. Mutation-based fuzzing is attractive because only few sample inputs (seeds) for the target program are required. The seeds are initially added to the collection of base inputs (corpus), from which mutated test inputs are generated. Mutation-based fuzzing is particularly effective when test inputs that trigger previously unseen behavior of the target are added to the corpus, called *feedback-driven fuzzing.* For coverage-guided fuzzing, the execution of new code paths or blocks is considered as previously unseen behavior.

Listing 1: Buggy function with possible stack overflow, inspired by [23].

```
void process_data(char* buffer, unsigned int length) {
  char stack_array[20];

  if( length > 0 && buffer[0] == 'b')
    if( length > 1 && buffer[1] == 'u')
      if( length > 2 && buffer[2] == 'g')
        if( length > 3 && buffer[3] == '!')
          memcpy(stack_array, buffer, length);
}
```

Consider the code of the function `process_data` in Listing 1 that causes a stack overflow when the first four characters of `input` match `"bug!"` and `length` has a greater value than 20. While a blackbox fuzzer needs to guess the first four characters correctly from $2^{8*4} = 2^{32}$ combinations at once, a coverage-guided fuzzer can progress on each comparison step with 2^8 possible combinations individually, increasing the overall probability of generating an input that triggers the stack overflow during fuzzing.

Typical user programs are fuzz tested by leveraging *source code instrumentation,* such that code coverage gets fed back to the fuzzer by additionally inserted code. Alternatively, *emulators* are used to obtain code coverage feedback when no source code, and therefore no code instrumentation at compile time, is available.

2.2 Coverage-Guided Fuzzing for Embedded Systems

Many approaches use *emulation* for fuzzing embedded systems. Emulators allow high transparency of the target execution. Gathering code coverage from an emulator is trivial and fuzzing in an emulator can be easily scaled. However, re-hosting embedded software into an emulator is an open research problem for decades [17, 58]. While simulating the microprocessor and its instruction set is feasible, it is emulating the exact behavior of *hardware peripherals* that remains challenging. If hardware peripherals are not emulated precisely, the emulated execution might diverge from an execution on real hardware, or even fail completely.

Several approaches have been presented to tackle the re-hosting problem. HALucinator [8] re-hosts embedded applications at the Hardware Abstraction Layer (HAL) based on the observation that HAL functions are somewhat device independent. Code that accesses the hardware directly, such as drivers, cannot be tested in this way, and the method requires manually-written *substitutions* for all hardware accessing functions. Avatar^2 [38] forwards all I/O requests from the emulator to the actual hardware device via a debugging interface, termed as *peripheral proxying*. Transferring data

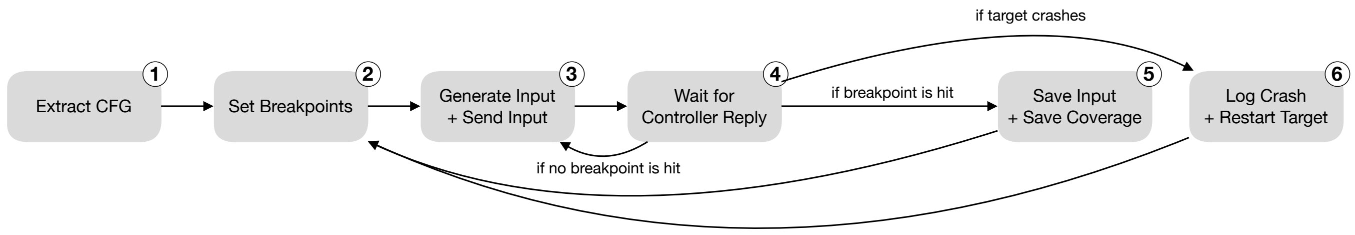
Extract CFG
1
Set Breakpoints
2
Generate Input + Send Input
3
Wait for Controller Reply
4
Save Input + Save Coverage
5
Log Crash + Restart Target
6
if target crashes
if breakpoint is hit
if no breakpoint is hit

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

between the emulator and the actual device introduces execution slowdowns of up to 80× [39] and requires a device-specific setup.

Recently, *peripheral modeling* has been proposed by Feng *et al.* [18], alongside their tool P^{2}IM, where the fuzzer is used to model hardware peripherals iteratively. Their idea is to first fetch input data from the fuzzer and then execute the embedded application in a simulator. For each occurring read on the I/O address space a portion of the fuzzing data is replied until the execution gets stuck, or the input buffer is exhausted. As a result, the fuzzer learns which values are required for further execution of the firmware, because it then achieves more code coverage. The idea has been adopted and refined in Jetset [28], μEmu [60], and Fuzzware [46]. Though *peripheral modeling* approaches are supposed to work out-of-the-box, it is unclear how transferable findings are. In our experience, more complex input interfaces like USB pose tremendous problems for peripheral modeling, as we show in our evaluation.

Other hardware-based approaches for coverage-guided fuzzing are applicable in specific settings only. Harzer Roller [4] injects function tracing and stack smashing detection into closed-source object files. This, however, lacks fine granular code coverage, and again requires the ability to change the program code. Boersig *et al.* [7] use source-code instrumentation and transfer the data via a debugging interface. Again, this requires the ability to change the program code, and is available only for ESP32 microcontrollers. Finally, μAFL [31] uses ARM's tracing mechanism Embedded Trace Macrocell (ETM) [54] with a conforming tracing hardware. Such tracing hardware, however, is expensive and rarely available.

2.3 Debuggers and Breakpoints

Debuggers are common tools to observe program executions, notably to understand unexpected program behavior. A popular debugger for user programs is GDB [33]. It enables to halt the execution of the target program on desired points, to examine memory values, and to single-step through the code. Microcontrollers feature different kinds of debug interfaces, such as a Joint Test Action Group (JTAG) port. Using such a port, *debug probes* have direct access to the hardware. Debug probes can implement the GDB *remote stub*, such that GDB can perform debug operations via the GDB *remote serial protocol* [21]. Typically, GDB runs on the programmer's PC that hosts the development environment and the source files of the application. From our observations, the GDB remote stub is implemented by most available on-board and off-board microcontroller debuggers [35], for instance from Segger [47], STMicroelectronics [49], or Lauterbach [30]. It can therefore serve as a generic way to gain insights into the execution of an embedded system.

Debug interfaces on commercial devices are ideally closed or disabled to prevent attacks. However, it has been shown several times in the past that disabled debug interfaces can be reopened, using fault injection attacks like power or clock glitching [29, 48]. Also, the firmware from a commercial device can be transferred to an equivalent development board with accessible debug interface, and thus enable debugging.

Debuggers use *breakpoints* to stop the execution at desired locations [57]. *Software breakpoints* are realized by replacing the original instruction in the software binary by a distinct instruction word that triggers an interrupt when executed. Upon resuming execution, the debugger re-inserts the original instruction. Software breakpoints therefore require rewriting small parts of the target program.

Continuous rewriting of memory can take time and can wear out the device's (flash) memory. If the program is stored on a read-only memory, such rewriting is impossible. Microcontrollers therefore usually feature a number of *hardware breakpoints.* Hardware breakpoints correspond to actual *registers* on the microprocessor and, once activated, interrupt program execution when the program counter value equals its register value. Hence, hardware breakpoints can be set to any program address, regardless of the memory type the respective code is stored in.

There are approaches that use *software breakpoints* for measuring code coverage and obtaining fuzzing feedback, in order to avoid the overhead and impediments of source code instrumentation [25, 40, 41]. The idea is to insert software breakpoints into unreached basic blocks and therefore allow the program to execute at full speed until new coverage is reached. Once the execution runs into a breakpoint, the corresponding instruction is removed from the binary to avoid further overhead. Oh *et al.* [44] use *software breakpoints* to measure code coverage in embedded firmware. They extract the start address of each basic block of the program during compilation and insert software breakpoints at each of them, also removing them once they are hit.

3 CONTROL FLOW GRAPHS

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

3.1 Dominator Relations of Control Flow Graphs

Dominator Relations describe further coherence between nodes in the a control flow graph. We use the notion of pre- and postdominator from Agrawal [1] and assume that control flow graph G has exactly one entry point and exactly one exit point[2].

Definition 1 (Predomination)**.** A node $u \in G$ *predominates* another node $v \in G$, denoted as $u \xrightarrow{pre} v$, if every path from the entry node to v contains u.

Definition 2 (Postdomination)**.** A node $w \in G$ *postdominates* another node $v \in G$, denoted as $w \xrightarrow{post} v$, if every path from v to the exit node contains w.

The dominator relations can be represented as (dominator) *trees* and be computed efficiently [9] for functions. The postdominator tree equals the predominator tree from the reversed control flow graph [1]. From any pre- and postdominator tree, we can derive the following transitive knowledge about other nodes:

Theorem 1 (Reachability)**.** If node v is reached, all parent nodes in the predominator tree have been reached before and all parent nodes in the postdominator tree will be reached afterwards.

3.2 Interprocedural Control Flow Graphs and Dominator Relations

Interprocedural control flow graphs describe possible transitions of basic blocks within whole programs instead of only functions. In principle, we could connect the control flow graphs from each function of the program (local CFGs) by adding all call and return instructions as edges. However, this introduces ambiguities when a function has multiple callers, because it creates paths from every calling function to every return point. When traversing the resulting interprocedural control flow graph to calculate a dominator graph, a return edges must only lead back to the actual call site of the current function, which requires context-sensitive algorithms [2, 10]. Published context-sensitive algorithms are complex and implementations are rarely available. GDBFuzz uses dominator relations as a bonus for reducing overhead, only, and therefore we developed the following simple approach.

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

We construct a *semi-interprocedural* control flow graph, where we connect the function control flow graphs by inserting all calls as edges from the call site to the callee. Return edges are omitted when building the semi-interprocedural CFG, so no incorrect flow can be introduced.

For the *reversed semi-interprocedural* control flow graph, we reverse the local control flow graphs, skip call edges, and only add the return edges. Again, we avoid inserting ambiguities by removing context-sensitive call edges. The corresponding semi-interprocedural dominator trees can then be calculated efficiently with the algorithms for local control flow graphs. Compared to full interprocedural dominator graphs, we might miss one dominance relation per call edge in the worst case, which should not really impair the fuzzing performance.

For convenience, we merge the pre- and postdominator graph: $\{(u, v)|u \xrightarrow{pre} v \vee u \xrightarrow{post} v\}$, requiring us to only handle a single dominator graph for the whole target program.

4 DESIGN

As shown in Figure 2, GDBFuzz leverages the control flow graph of the target program to set available hardware breakpoints to randomly chosen basic blocks that are yet unreached. It then repeatedly generates test cases by applying mutations to randomly chosen inputs from the corpus, and sends the test cases to the target input interface. If the debug probe signals a breakpoint hit, GDBFuzz marks the corresponding node and its dominating nodes as reached, and adds the responsible test case to the corpus. Test cases that cause crashes or timeouts are preserved separately. When no breakpoint interrupt occurred after a predefined amount of exercised test cases, GDBFuzz relocates the hardware breakpoints to newly chosen nodes. After each relocation, GDBFuzz first tests all inputs from the corpus again to check if they already reach the newly targeted basic blocks. Like coverage-guided fuzzing with full code instrumentation, the evolutionary algorithm causes the input corpus to grow over time with inputs that reach different code areas.

The remainder of this section describes how we extract the CFG, how we find a fuzzing entry point in the target application, and how we detect and handle bugs during execution.

4.1 Extracting the Control Flow Graph

Control flow graphs can be obtained trivially during program compilation, because the compiler is aware of the whole control flow. However, GDBFuzz is designed to work on binaries to broaden its applicability, as source code may not be available for all software components on an embedded system. *Ghidra* [42] is an open source reverse engineering tool which supports most common processor architectures, is scriptable, and is therefore well suited for our needs. Like all binary disassembling approaches, *Ghidra* cannot guarantee to detect all control flows, especially when it comes to indirect branches or aggressive compiler optimizations [45]. Therefore, we refine and update the control flow graph iteratively during fuzzing, which we describe in Section 5.

4.2 Finding the Entry Point

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

(1) Send a test input to the target device and interrupt the execution immediately.
(2) Use `gdb find` to rediscover the sent input in the memory of the devices.
(3) Set a data watchpoint to the first address of the rediscovered input data.
(4) Send the test input again.
(5) All program counter addresses on now occurring interrupts are candidates for an entry point.

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

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

4.3 Detecting Bugs

Bug oracles detect whether a bug is triggered during execution. Since fuzzing origins from testing user applications, a common bug oracle is to observe the target process on raised error signals, e.g. segmentation faults. To find bugs that do not trigger faults directly, sanitizers and assertions are used. These are usually deployed at compile time, but there are methods to inject sanitizers directly into binaries [11]. However, more sophisticated bug oracles are still an open research problem [5] and out of scope of this work.

GDBFuzz relies on the triggered bugs being observable, meaning that faults or other misbehavior must be triggered by the bug. Silent corruptions, as demonstrated in [39], can therefore not be discovered, unless additional sanitizers are used during compilation. Faults can be detected, for instance, by occurring connection errors like timeouts or error response codes. Additionally, breakpoints can be set on locations of fault handlers via the debugging interface, which also catches fault signals from deployed sanitizers.

Since the location of the fault handlers in the code usually does not change during runtime, software breakpoints can be used to detect their execution such that all hardware breakpoints are available for the coverage feedback mechanism. Software breakpoints are well suited in this case, since they are not repositioned during fuzzing. If software breakpoints are not available for the System under Test (SuT), a subset of available hardware breakpoints can be used, too, with the disadvantage of decreased fuzzing performance.[3]

Obtaining the locations of fault handlers is done in a manual to semi-automated way, because embedded systems vary dramatically in features, such as processors, operating systems, frameworks, libraries, and sanitizers. In our experience, the default fault handlers e.g. from freeRTOS [20], Arduino [3], and STM32CubeMX [53] typically end in an infinite loop. *Ghidra* can identify functions with infinite loops [43], which can then be considered as potential fault handlers. For all our test applications, it was sufficient to rely on timeouts that are provoked by the infinite loops in the fault handlers, not requiring any further setup work for us.

[3]We evaluate the influence of the number of available breakpoints on performance in Section 6.

4.4 Handling Bugs

Whenever GDBFuzz detects a crash or a timeout, it

(1) *deduplicates* the bug to identify whether the bug is *unique*, i.e. whether it is the first time that this bug was found;
(2) *preserves* the input triggering this bug if the bug is unique;
(3) *restarts* the target system; and
(4) *continues* fuzzing.

The same bug may be triggered multiple times during fuzzing. Analyzing each bug requires substantial efforts, which is why deduplication is required. The goal is to provide the test engineer a minimal set of inputs triggering only unique bugs. We use hashes of the call stack [34] to uniquely identify and deduplicate bugs.

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

5 IMPLEMENTATION

GDBFuzz consists of the following components:

Test Data Generator. Like all coverage-guided fuzzers, GDBFuzz preserves inputs that trigger different code areas in the input corpus, and derives new inputs by mutating these. Dozens of general purpose mutation-based fuzzers have been published in recent years [34]. We therefore do not develop a mutation algorithm from scratch, but reuse the mutation engine from libFuzzer [32]. The actual mutation engine in GDBFuzz is easily interchangeable.

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

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

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

Table 1: Details of our development boards including architecture, utilized debug probe, and the number of available hardware breakpoints.

Board	Arch.	Debug Probe	#HW Br.
STM32L4S5I [50]	ARM	STLinkv3	6
CY8CKIT-062-WIFI-BT [27]	ARM	KitProg3	6
ESP32-DevKitC_V4 [16]	Xtensa	J-Link Ultra	2
EXP430F5529LP [55]	MSP430	eZ-FET lite	8

Table 2: Application classes and initial input seeds for our case study on embedded hardware[5].

Name	Description	Seed
Buggy	Buggy program from Listing 1	None
JSON	Parses serial data as *json* string	"1000, 2000, 3000"
USB	USB mass storage client	`55 53 42 42 00...`[4]
HTTP	HTTP server via WiFi	"GET / HTTP1.1"

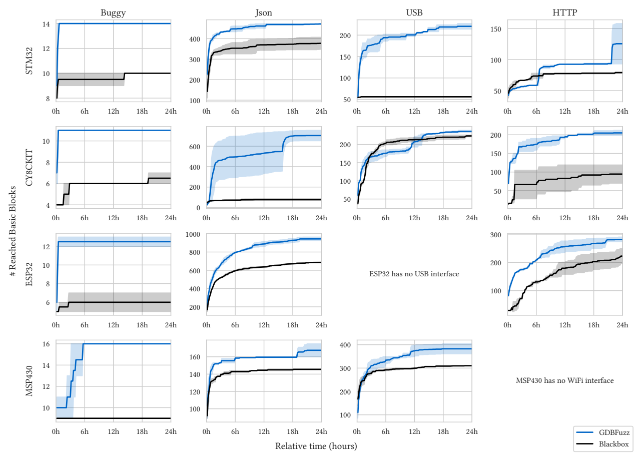

Figure 3: Reached basic blocks over time for GDBFuzz and blackbox fuzzing on embedded hardware (N=2).

Dynamic Control Flow Graph Refinement. As mentioned earlier, reverse engineering tools cannot guarantee to detect the whole control flow of a program [45]. Missing control flow manifests itself as a dangling node in the CFG without successor that is not marked as *terminal* by *Ghidra*. When finding a test input that triggers the execution of such a dangling node, we perform the following steps:

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

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

6 EVALUATION

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

(1) For the hardware-based setting we pick a variety of common development boards, listed in Table 1 together with their corresponding architectures, utilized debug probes, and the number of available hardware breakpoints. On each development board we deploy four different classes of applications, listed in Table 2 with an initially given seed. The *Buggy* program exposes the buggy function from Listing 1 to a serial input interface and serves as a ground truth. The specific applications for each board are derived from examples shipped with the development boards, or compatible toolchains[5]. The *HTTP* and *USB* application classes require the appropriate interface to exist on the target board.
(2) The application-based setting features 16 programs from Google's Fuzzer Test Suite [24], compiled as *x86* linux applications, and provides a scalable and independently measurable evaluation environment. GDBFuzz can execute an application either in a QEMU instance, enabling live measuring of reached code coverage, or with GDB directly, enabling low overhead and unlimited amounts of breakpoints[6]. We compile the applications with compiler optimizations (`-O3`), and execute the corresponding experiments on a server with four Intel Xeon Gold 6144 CPU's and 1.48TB of RAM.

6.1 GDBFuzz vs. Blackbox Fuzzing

GDBFuzz enables coverage-guided fuzzing on systems where coverage measurement is hardly possible[7]. We therefore utilize the partial coverage extraction mechanism of GDBFuzz itself, to measure coverage differences between GDBFuzz and blackbox fuzzing in our hardware-based setting. Specifically, we deploy GDBFuzz, but omit adding new inputs to the corpus during fuzzing to simulate a blackbox fuzzer. As a result, we can investigate how the evolutionary fuzzing algorithm of GDBFuzz performs, and can consequently address our first research question:

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

Figure 3 shows coverage over time plots for all board and application class combinations. Each experiment is repeated twice, leading to an accumulated experiment time of 56 days. Without exception, GDBFuzz achieves a higher code coverage across all runs than blackbox fuzzing and shows that it can greatly benefit from the partial coverage information it retrieves via hardware breakpoints. In particular for the *Buggy* program, blackbox fuzzing has little to no chance to fulfill all conditions to trigger the contained stack overflow bug, as theoretically described in Section 2.1. In this application class, GDBFuzz achieves almost 100 iterations per second on the powerful *CY8CKIT* board, while it can only reach about 1.5 iterations per second on the low performance *MSP430* board. This explains why it takes way longer for GDBFuzz to solve the input constraints on the latter and we can also see how important

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

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

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

throughput is for fuzzing. Nevertheless, GDBFuzz performs well on all of our development boards, finds the bug in all cases, and reports the resulting crashes properly.

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

6.2 GDBFuzz vs. State of the Art

As mentioned in Section 2.2, there are multiple approaches that claim to enable coverage-guided fuzzing for embedded systems, raising the following question:

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

μAFL [31] is a hardware-based embedded fuzzer that uses the ARM ETM interface to extract code coverage from an embedded program. Our *CY8CKIT-062-WIFI-BT* development board features such an ETM interface, and we have access to the tracing hardware required therefore. However, the publicly available version of μAFL reported implausible results on our setup, which we could not resolve despite having vendor support. We noted that μAFL uses raw trace data functions, which are unreliable in some implementations, and are marked for *internal use only* [22]. Also, and in contrast to GDBFuzz, μAFL requires very specific hardware, which is why it is not a direct competitor; we are not aware of a generic hardware-based embedded fuzzing approach to compare GDBFuzz against.

Most of published embedded fuzzing methods are emulation-based, from which only *peripheral modeling* approaches can enable coverage-guided fuzzing for embedded systems on a scale and are competitors to GDBFuzz. We therefore compare GDBFuzz against the latest peripheral modeling approach Fuzzware [46], whose authors claim embedded fuzzing on the actual hardware to be impractical. Fuzzware works on all *ARM Cortex-M*-based microcontrollers, so we can fuzz all applications from the first two development boards in Table 1.

First, we need to agree on how we compare emulation-based to hardware-based approaches. Li *et al.* [31] compared *peripheral modeling* approaches to their hardware-based approach μAFL by the number of the achieved fuzzing iterations per hour. We agree that the number of executions per time is an important metric for fuzzing. However, for a fair comparison, the same or at least similar code areas must be executed in that time. Peripheral modeling approaches like P^2IM [18] and Fuzzware [46] use fuzzing to iteratively carve an artificial execution environment for the firmware. Over time the peripheral models are refined and the execution speed decreases since the firmware can be further executed. By design, peripheral modeling does not target specific code areas. This makes throughput a meaningless measure for comparing these different approaches, because it is unclear whether the same code parts are executed in this time.

We therefore compare embedded fuzzing approaches based on the number of reached basic blocks in a targeted region of the firmware during fuzzing, as also done in [46].

Table 3: Covered basic blocks by Fuzzware with 16 cores, and GDBFuzz after 24 hours of fuzzing.

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

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

Table 3 lists the absolute and relative number of reached basic blocks using Fuzzware and GDBFuzz. Although it had significantly more computing power provided, Fuzzware did not reach *any* basic block on six out of eight applications, whereas GDBFuzz covered a substantial part of them. On the remaining two applications, GDBFuzz reached more basic blocks than Fuzzware. The USB controller on the STM32 board transfers data via Direct Memory Access (DMA), which is not supported by Fuzzware, but is required to execute the application. From our experience DMA is a widely used mechanism to interact with hardware peripherals, and the lack of DMA support by Fuzzware is a major drawback. The *WiFi* and Transmission Control Protocol (TCP) protocol handling on the STM32 board takes place in a separate chip connected via Serial Peripheral Interface (SPI) to the microcontroller. In order to trigger the execution of the HTTP parser, Fuzzware would need to model the inter chip communication protocol correctly, which it did not.

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

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

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

6.3 Reveal Bugs with GDBFuzz

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

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

For answering **RQ3**, we first have a look at two known real-world bugs. The USB enumeration handling of the STM32CubeL4 USB Middleware [51] contains the known vulnerabilities *CVE-2021-34259* and *CVE-2021-34268* that were found using μAFL [31]. We verify that GDBFuzz can detect such real-world vulnerabilities and use a firmware based on the USB host mass storage device class (MSC) example application version 1.17.1 from STM32CubeL4 [52]. The STM32 microcontroller acts as USB host in this MSC application. Our setup therefore is similar to that from the μAFL authors. To generate USB traffic, we plug a common USB flash drive as USB client into the USB port. We then introduce a fuzzing harness into specific stages of the USB enumeration, where we replace the USB data frame with fuzz data just before the USB host processes this data. Namely, we replace the raw *device descriptor* or the *device configuration* that is sent by the client device before any parsing of the fuzz data. The introduced fuzzing harness receives fuzz data from GDBFuzz via a serial interface. Both mentioned *CVEs* manifest themselves as timeout when triggered, because the USB host middleware gets wrongly configured by malformed *device descriptor* or *device configuration* USB packets. Basically they are caused by missing validity checks for the untrusted data from the USB client. With GDBFuzz and the appropriate fuzzing harnesses, both *CVEs* are triggered and detected in less than 5 minutes during our experiments.

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

(1) An infinite loop in the STM32 USB device stack, caused by counting a `uint8_t` index variable to an attacker controllable `uint32_t` variable within a *for loop* [13].
(2) A buffer overflow in the Cypress JSON parser, caused by missing length checks on a fixed size internal buffer [12].
(3) A null pointer dereference in the Cypress JSON parser, caused by missing validation checks [14].

GDBFuzz reveals real vulnerabilities in embedded software.

6.4 GDBFuzz vs. AFL++

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

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

For a fair comparison between GDBFuzz and AFL++ [19], we let them operate on the uninstrumented binary using QEMU mode for AFL++ (`-Q`) and GDBFuzz with QEMU, too. As live measurement is impossible with the modified QEMU version included in AFL++, we replay the respective input corpora after fuzzing and measure the reached number of basic blocks thereby. We configure GDBFuzz to use eight breakpoints, which is a realistically low number of breakpoints available in real microcontrollers.

Figure 4 shows coverage over time plots for GDBFuzz and AFL++. Obviously, AFL++ covers more code over time than GDBFuzz.

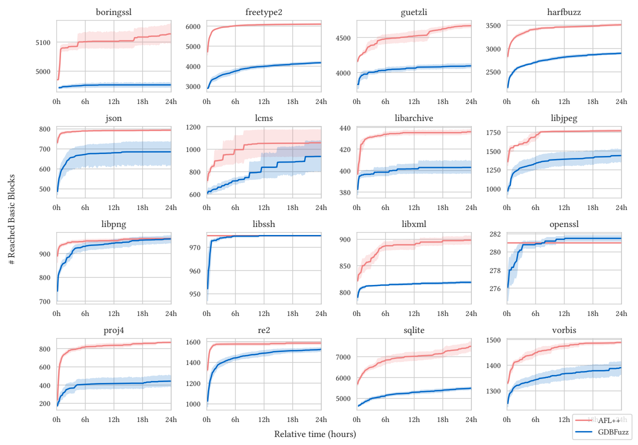

Figure 4: Basic block covered by GDBFuzz and AFL++ on applications where code instrumentation is possible (N = 10).

AFL++ is designed and optimized for exactly these kind of applications and can benefit from its exhaustive code instrumentation. However, we think that GDBFuzz with just eight utilized breakpoints is not too far away. On some application, GDBFuzz could even reach a similar number of basic blocks. We also emphasize that AFL++ falls back to *blackbox fuzzing* in scenarios where emulation and instrumentation is not available—and this is again where GDBFuzz is superior.

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

6.5 Boost by Dominance Relations

Let us now evaluate specific elements of the GDBFuzz design. As described in Section 4, GDBFuzz makes use of *dominance relations* to mark multiple basic blocks as reached with a single breakpoint interrupt, which leads to the question:

RQ5: How much does GDBFuzz benefit from dominance relations?

To answer **RQ5**, we analyze the average number of breakpoint interrupts, as well as the average number of reached basic blocks during the previous experiments in Table 4.

Across all our experiments, each breakpoint interrupt led to 3.15 marked basic blocks on average, meaning that the number of probed basic blocks is reduced by 68.25%. This ratio is better than in experiments of the efficient code instrumentation algorithm presented in [56], where the authors achieved to reduce the number of instrumentation points only by 34% to 49% in their experiments. GDBFuzz can presumably reduce the overhead further, because we additionally use post dominance relations and the described semi-interprocedural CFG.

As we can see in the *Precision* column, the vast majority of the marking dominating basic blocks was correct. Incorrectly marked basic blocks can result from incorrect reverse engineered control flow. The reached precision of mostly more than 99% is sufficient for coverage-guided fuzzing since it is a stochastic process and

Table 4: Averaged results of the benchmark (N=10).

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

does not rely on 100% correct coverage data. Popular fuzzing tools, like AFL++, store coverage data in hash maps and also miss some coverage during fuzzing due to hash collisions.

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

6.6 Revealing Control Flow

GDBFuzz can guide reverse engineering tools to reveal undetected control flow, as we described in Section 5 and we investigate by the question:

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

We answer **RQ6** by compareing the average relative number of additional revealed basic blocks and edges against the ones that *Ghidra* initially detects. Since the targets have been compiled with activated compiler optimizations, recovering the control flow is particularly hard for reverse engineering tools. In Table 4, we can see that up to 674.16% additional basic blocks and 708.61% additional edges could be revealed during our experiments. A lower number of newly found control flow does not necessarily show a lower performance from GDBFuzz, but rather a good reverse engineering performance of *Ghidra*.

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

6.7 Number of Available Breakpoints

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

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

For estimating how different numbers of breakpoints influence the fuzzing performance, we execute the applications directly with GDB and use ordinary software breakpoints for the feedback, since

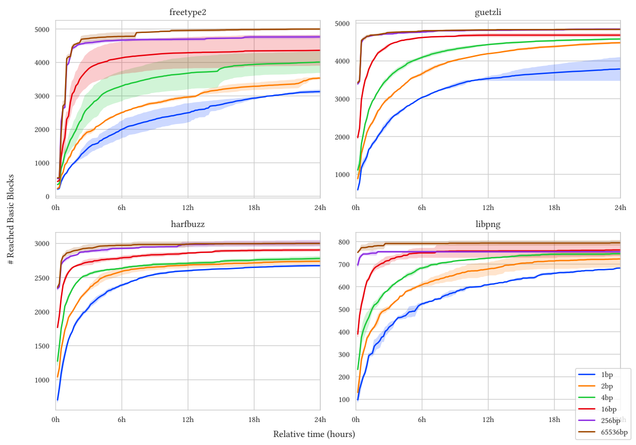

Figure 5: Fuzzing performance of GDBFuzz on four applications using different numbers of virtual breakpoints (N=2).

QEMU does not scale well with an increasing amount of breakpoints. This way we have arbitrarily many breakpoints available without affecting execution time, and can estimate how their number affects the achieved coverage over time. To answer **RQ7**, we execute GDBFuzz in the application-based setting using an exponentially increasing number of virtual breakpoints from 1 to 65536. Representative, Figure 5 shows the reached basic blocks over time on four applications[8], averaged from 2 runs.

Unsurprisingly, more used breakpoints lead to more covered code blocks per time. In our experiments, it roughly seems that doubling the number of breakpoints yields to a linear improvement of fuzzing performance. Exponential correlations between effort and revenue are common in the research area of fuzzing [6]. Likewise, our experimental observation between the number of utilized breakpoints and coverage over time suggests an exponential correlation.

Linearly more coverage over time requires exponentially more breakpoints.

6.8 GDBFuzz vs. Blackbox Fuzzing

As pointed out in Section 6.4, AFL++ falls back to *blackbox fuzzing* when no conforming instrumentation mechanism is available. This motivates our final research question:

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

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

Figure 6 shows measured coverage over time results for blackbox fuzzing, GDBFuzz, and a trimmed version (GDBFuzz$_{\text{Simple}}$) that does not make use of dominance relations as described in Section 3. This larger scale benchmark with independent code coverage measurements confirms the results from the development boards:

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

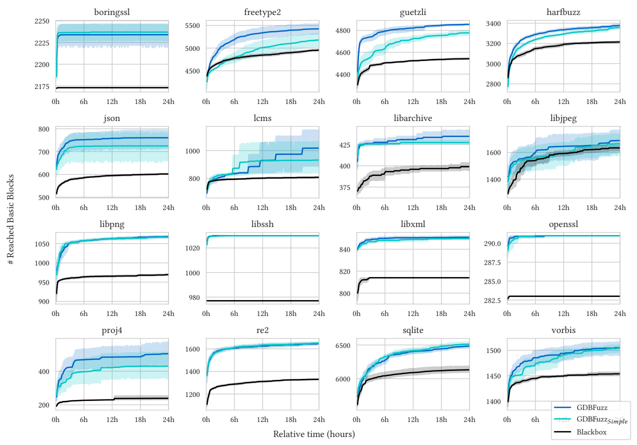

Figure 6: Reached coverage from GDBFuzz with and without using dominance relations, and blackbox fuzzing measured by QEMU (N = 10).

GDBFuzz outperforms blackbox fuzzing in all experiments. Furthermore, we can see that using dominance relations to gain transitive knowledge led to more and faster code coverage on most target applications. Nevertheless, even GDBFuzz$_{\text{Simple}}$ greatly outperforms blackbox fuzzing in all experiments.

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

7 DISCUSSION

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

7.1 Block and Branch Coverage

While most state-of-the-art fuzzers like AFL++ [19] and libFuzzer [32] leverage edge coverage, we use block coverage to guide the evolutionary fuzzing algorithm. Extracting edge coverage with our methods, would require to probe already reached blocks multiple times, which would increase overhead drastically. Nagy *et al.* [40] use software breakpoints to detect the execution of new basic blocks for normal software. They find that edge coverage cannot benefit from its finer granularity because of the required constant instrumentation overhead. Since edge instrumentation would introduce even more overhead in our setting than source code instrumentation on normal software, we estimate block coverage as the only feasible coverage metric in GDBFuzz.

7.2 Fuzzing Firmware Drivers

Fuzzing drivers, or the middleware of embedded software, can be implemented in a blackbox approach, by injecting fuzz data to external facing interfaces, or as a whitebox approach, by compiling a fuzz harness into the firmware to fetch and redirect the fuzz data to the driver function. Whitebox approaches offer more flexibility since driver functions can be called directly and an acknowledgment signal can be fed back to the fuzzer, which indicates a function has properly returned. However, expert knowledge about the code and the system is required to implement a suitable fuzzing harness. Also, an implemented harness works on a distinct code base only, and false positives can be produced since input can be sanitized in the hardware already before the tested driver function is reached [31]. A whitebox approach was suitable for replicating the known *CVE's*, revealed by μAFL, since the faulty functions were known and the original finders have chosen the same way.

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

7.3 Threats to Validity

Empirical studies are necessarily fraught with threats to validity. To address *external validity* doubts, we have tested the method on different development boards with different debuggers and architectures and made sure that we can find real-world bugs. Additionally, we conducted a larger case study on known fuzzer benchmarking targets. To minimize the risk of systematic errors and addressing *internal validity* doubts, we decoupled coverage measurements from our tool during evaluation and repeated each experiment multiple times. The implementation of GDBFuzz is publicly available to allow reproduction of our results (Section 9).

8 CONCLUSION AND FUTURE WORK

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

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

Future work on GDBFuzz will focus on enhanced strategies for choosing basic blocks to probe and incorperating established fuzzing optimizations, like *Corpus Minimization, Dictionaries,* or different *Seed Schedules.* Also, fuzzing for stateful embedded systems is part of future work.

9 DATA AVAILABILITY

The open source implementation of GDBFuzz, the evaluation setup, and raw results are available at

https://github.com/boschresearch/gdbfuzz

ACKNOWLEDGMENTS

This work was funded by the German Federal Ministry of Education and Research (BMBF, project CPSec - 16KIS1565 and 16KIS1564K)

REFERENCES

[1] Hiralal Agrawal. 1994. Dominators, super blocks, and program coverage. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 25–34.

[2] Hira Agrawal. 1999. Efficient coverage testing using global dominator graphs. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 11–20.

[3] Arduino. 2022. Arduino Default Fault Handler. (2022). https://github.com/arduino/ArduinoCore-samd/blob/104f07f9053c49f60ceb0b09f9ae958fdee82cb8/cores/arduino/cortex_handlers.c#L28

[4] Katharina Bogad and Manuel Huber. 2019. Harzer roller: Linker-based instrumentation for enhanced embedded security testing. In *Proceedings of the 3rd Reversing and Offensive-oriented Trends Symposium*. 1–9.

[5] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Softw.* 38, 3 (2021), 79–86.

[6] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 713–724.

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

[8] 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.

[9] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. 2001. A simple, fast dominance algorithm. *Software Practice & Experience* 4, 1-10 (2001), 1–8.

[10] Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. 2007. A practical interprocedural dominance algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 4 (2007), 19–es.

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

[12] Max Eisele. 2022. Buffer Overflow in cy_json_parser.c. (2022). https://github.com/Infineon/connectivity-utilities/issues/2

[13] Max Eisele. 2022. Infinite Loop in STM32 SCSI Driver. (2022). https://github.com/STMicroelectronics/STM32CubeL4/issues/69

[14] Max Eisele. 2022. Null Pointer Dereference in cy_json_parser.c. (2022). https://github.com/Infineon/connectivity-utilities/issues/1

[15] Max Camillo Eisele, Marcello Maugeri, Rachna Shriwas, Christopher Huth, and Giampaolo Bella. 2022. Embedded Fuzzing: a Review of Challenges, Tools, and Solutions. *Cybersecurity* (2022).

[16] Espressif. 2016. ESP32-DevKitC V4. (2016). https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html

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

[18] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium (USENIX Security 20)*. 1237–1254.

[19] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*. 10–10.

[20] freeRTOS. 2022. Debugging Hard Fault & Other Exceptions. (2022). https://www.freertos.org/Debugging-Hard-Faults-On-Cortex-M-Microcontrollers.html

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

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

[23] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44.

[24] Google. 2016. Fuzzer Test Suite. (2016). https://github.com/google/fuzzer-test-suite Accessed: 2022-11-22.

[25] Samuel Groß. 2020. TrapFuzz. (2020). https://github.com/googleprojectzero/p0tools/tree/master/TrapFuzz

[26] Steve Heath. 2002. *Embedded systems design.* Elsevier.

[27] Infineon. 2018. CY8CKIT-062-WiFi-BT PSoC 6 WiFi-BT Pioneer Kit. (2018). https://www.infineon.com/cms/en/product/evaluation-boards/cy8ckit-062-wifi-bt/

[28] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. 2021. Jetset: Targeted firmware rehosting for embedded systems. In *30th USENIX Security Symposium (USENIX Security 21)*. 321–338.

[29] Sultan Qasim Khan. 2020. Microcontroller Readback Protection: Bypasses and Defenses. (2020).

[30] Lauterbach. [n. d.]. Lauterbach Development Tools. ([n. d.]). https://www.lauterbach.com Accessed: 2022-11-22.

[31] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. 2022. μAFL: non-intrusive feedback-driven fuzzing for microcontroller firmware. In *Proceedings of the 44th International Conference on Software Engineering*. 1–12.

[32] LLVM. 2015. libFuzzer – a library for coverage-guided fuzz testing. (2015). https://llvm.org/docs/LibFuzzer.html Accessed: 2022-11-22.

[33] Mike Loukides and Andy Oram. 1996. Getting to know GDB. *Linux Journal* 29 (1996).

[34] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).

[35] Eric Mercer and Michael Jones. 2005. Model checking machine code with the GNU debugger. In *Model Checking Software: 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005. Proceedings 12*. Springer, 251–265.

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

[37] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[38] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar 2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, Vol. 18. 1–11.

[39] 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*.

[40] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 787–802.

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

[42] National Security Agency. 2019. Ghidra. (2019). https://ghidra-sre.org/ Accessed: 2021-12-20.

[43] National Security Agency. 2022. Ghidra function hasNoReturn. (2022). https://ghidra.re/ghidra_docs/api/ghidra/program/model/listing/Function.html#hasNoReturn()

[44] JinSeok Oh, Sungyu Kim, Eunji Jeong, and Soo-Mook Moon. 2015. OS-less dynamic binary instrumentation for embedded firmware. In *2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII)*. IEEE, 1–3.

[45] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 833–851.

[46] 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. *31st USENIX Security Symposium (USENIX Security 22)* (Aug. 2022). https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski

[47] Segger. [n. d.]. Segger Debug & Trace Probes. ([n. d.]). https://www.segger.com/products/debug-trace-probes/ Accessed: 2022-11-22.

[48] Sergei Skorobogatov. 2011. Fault attacks on secure chips. *Design and Security of Cryptographic Algorithms and Devices* (2011).

[49] STMicroelectronics. [n. d.]. STLINK-V3 modular in-circuit debugger and programmer for STM32/STM8. ([n. d.]). https://www.st.com/en/development-tools/stlink-v3set.html Accessed: 2023-02-216.
[50] STMicroelectronics. 2020. B-L4S5I-IOT01A Discovery kit for IoT. (2020). https://www.st.com/en/evaluation-tools/b-l4s5i-iot01a.html
[51] STMicroelectronics. 2020. STM32 USB Host Library. (2020). https://github.com/STMicroelectronics/STM32CubeL4/tree/v1.17.1/Middlewares/ST/STM32_USB_Host_Library
[52] STMicroelectronics. 2021. STM32 USB MSC Standalone. (2021). https://github.com/STMicroelectronics/STM32CubeL4/tree/v1.17.1/Projects/B-L475E-IOT01A/Applications/USB_Host/MSC_Standalone
[53] STMicroelectronics. 2022. Cortex-M4 Fault Handler. (2022). https://github.com/STMicroelectronics/STM32CubeF4/blob/4aba24d78fef03d797a82b258f37dbc84728bbb5/Projects/STM32F411RE-Nucleo/Examples_LL/SPI/SPI_OneBoard_HalfDuplex_DMA/Src/stm32f4xx_it.c#L59
[54] Neal Stollon. 2011. ARM ETM. In *On-Chip Instrumentation*. Springer, 213–218.
[55] Texas Instruments. 2013. MSP430F5529 USB LaunchPad development kit. (2013). https://www.ti.com/tool/MSP-EXP430F5529LP
[56] Mustafa M Tikir and Jeffrey K Hollingsworth. 2002. Efficient instrumentation for code coverage testing. *ACM SIGSOFT Software Engineering Notes* 27, 4 (2002), 86–96.
[57] Bart Vermeulen. 2008. Functional debug techniques for embedded systems. *IEEE Design & Test of Computers* 25, 3 (2008), 208–215.
[58] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. 2021. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys (CSUR)* 54, 1 (2021), 1–36.
[59] Joobeom Yun, Fayozbek Rustamov, Juhwan Kim, and Youngjoo Shin. 2022. Fuzzing of Embedded Systems: A Survey. *ACM Comput. Surv.* (may 2022). https://doi.org/10.1145/3538644
[60] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic firmware emulation through invalidity-guided knowledge inference. In *USENIX Security Symposium*. 2007–2024.

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