# Hardfuzz: DataFlow-Guided On-Device Fuzzing for Microcontrollers (Registered Report)

Kai Feng
School of Computing Science
University of Glasgow
Glasgow, UK
k.feng.2@research.gla.ac.uk

Jeremy Singer
School of Computing Science
University of Glasgow
Glasgow, UK
jeremy.singer@glasgow.ac.uk

Angelos K Marnerides
Dept. of Electrical & Computer Engineering
KIOS CoE
University of Cyprus
Nicosia, Cyprus
marnerides.angelos@ucy.ac.cy

***Abstract*—Fuzzing firmware on microcontrollers (MCUs) is hard to scale. Rehosting is an ideal way to achieve this, but it often loses fidelity and can be slow, while on-device tracing support is limited. Standard coverage-guided fuzzing relies on software instrumentation, which is costly for MCUs and gives only control-flow signals that arrive late for complex checks.**

**We present Hardfuzz, an on-device fuzzer that uses *definition-use (def-use)* chains to guide exploration. Hardfuzz performs offline static analysis to extract def-use pairs from the binary, then runs directly on the device and uses the debug unit's hardware breakpoints to observe when definitions and their uses execute. Two small bitmaps in shared memory record (i) which definitions execute and (ii) which def-use pairs execute, giving rich feedback than basic-block coverage alone. A lightweight scheduler prioritises definitions with many uses and adapts to the few hardware breakpoints available on MCUs.**

**We evaluate Hardfuzz against another hardware breakpoint-based solution, GDBFuzz. In emulation, Hardfuzz achieves higher basic-block coverage in most targets and progresses faster in the early hours running on emulation. On hardware, it covers 14-40% more basic blocks after 24 hours across three programs with known faults. These results show that def-use guidance is practical on MCUs and improves exploration over control-flow-only feedback.**

## I. Introduction

Fuzzing microcontroller (MCU) firmware remains difficult to scale and automate. MCU firmware is tightly coupled to board-specific peripherals via memory-mapped I/O (MMIO), DMA, and interrupts [1], hindering direct execution on a host and complicating high fidelity rehosting [2]. Hardware-in-the-Loop (HiL) can improve realism by interacting with real devices [3]–[5], but introduces synchronisation and I/O bridging overhead that decreases throughput, which is an issue for feedback-driven fuzzing campaigns.

This exposes a central tension between *fidelity* and *execution speed*. Rehosting can accelerate input throughput but may diverge from real hardware behaviour when peripheral models are incomplete or slow to evaluate. On-device execution provides ground truth with maximum fidelity, yet incurs per-test overheads from debug halts, breakpoint churn, and probe latency. HiL typically sits between these extremes: higher fidelity than pure emulation but slower than ideal due to cross-boundary communication.

```
1  int process_packet(const uint8_t *in,
       size_t n) {
2      if (n < 8) return -1;
3      uint16_t len = (uint16_t)((in[0] << 8)
        | in[1]);          /* (D1) */
4      if (len > n - 4) return -2;
5      const uint8_t *payload = in + 4;
6      uint32_t token = crc32_like(payload,
       len) ^ 0x5A5A5A5AA;          /* (D2) */
7      if (((token ^ 0xA5A5A5A5A) & 0x3u) !=
       0)                  /* (U1) */
8          return 0;
9      if ((((token >> 8) + len) % 29u) != 7u
       )                  /* (U2) */
10         return 0;
11     uint8_t buf[128];
12     memcpy(buf, payload, len);  /* (U3) */
13     return 1;  /* deep path reached */
14 }
```

Listing 1. Example where def-use guidance provides earlier signals than basic-block coverage. (D1,D2) are definitions; (U1,U2,U3) are uses.

A second challenge is the *quality of feedback*. The canonical feedback signal, control-flow (CF) coverage from software instrumentation is often impractical on MCUs due to tight memory and timing budgets [6], [7]. Even when feasible, CF coverage is coarse: it only reveals *where control went* (which blocks/edges executed), not *what values were computed*. After a fuzzer learns to reach the blocks containing early checks, many subsequent mutations still execute the *same* blocks while merely changing internal values. Thus CF coverage quickly *saturates*: the fuzzer receives no new reward even if it is making progress toward satisfying deep value constraints. Two inputs can look identical under CF coverage whether the computed value is "one off" or far away.

We instead track *definition-use (def-use)* chains: edges from where a value is defined (D1/D2) to where that *same dynamic value* is later used (U1–U3) along the executed path. Observing that a particular *use* consumed the value from a particular *definition* yields a richer, earlier signal than CF

Fuzzing Workshop (FUZZING) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-04-6
https://dx.doi.org/10.14722/fuzzing.2026.23015
www.ndss-symposium.org

coverage. Intuitively, CF says *"did we reach this check?"*; DU also says *"which computed value fed this check?"*. This extra resolution produces intermediate milestones: the fuzzer can be rewarded when the derived `token` (or `len`) reaches progressively deeper uses (U1 $\rightarrow$ U2 $\rightarrow$ U3), instead of waiting until the deepest block is finally executed.

Listing 1 shows a common pattern. A CF-guided fuzzer can reach the blocks containing checks (U1, U2). Thereafter it receives no new reward until the deepest block is reached (U3). It must rely on random mutations to hit exact values. DU coverage, in contrast, provides an earlier gradient: reaching U2 yields a new $(D2 \rightarrow U2)$ event even though the executed blocks may be unchanged, which marks genuine progress toward the deep path. This point has also been proved in prior work [8]–[10].

Thus, we present **Hardfuzz** which extracts def-use chains offline from the firmware and uses the debug unit's hardware breakpoints to detect, on device, when definitions and their uses are executed. Two compact bitmaps in shared memory record (i) definition hits and (ii) def-use hits. A comparator-aware scheduler prioritises definitions with many uses and sets uses in proximity-ordered chunks after a def hit, respecting the few hardware comparators typical of MCUs. In this way, the def-use chain will reflect the change of the dataflow coverage and provide feedback for fuzzing campaign.

The contributions of the paper are shown as below:

- **On-device, def-use-guided fuzzing.** We show that def-use chains can guide fuzzing on commodity MCUs using only the debug unit.
- **Comparator-aware orchestration.** We set definitions first and then nearby uses in chunks, fitting small hardware comparator budgets without stalling execution.

## II. Technical Background

### A. Fuzzing

Embedded software couples firmware tightly to peripherals and often executes on bare metal or a small RTOS across diverse architectures. Fuzzing mutating inputs to trigger unexpected behaviours has a strong track record in conventional software but remains challenging for firmware because fidelity (accurate device interaction) and coupling (architectures/peripherals) complicate harnessing and feedback, there is no "one-size-fits-all" solution [11].

**Core components.** Across execution targets (hardware, emulation, HiL), fuzzers share four steps: (i) *seed selection*, (ii) *mutation/scheduling*, (iii) *feedback* (coverage/fitness), and (iv) *bug detection/triage*. Seeds may come from traces (protocols/files) or from minimal bootstraps with harness prologues for stateful handshakes [12]–[14]. Corpus growth/minimisation follows AFL-style practices [7]. Mutations range from byte-level operators to domain-aware tweaks for sensor/packet fields and sequence-aware strategies for protocols; schedulers adapt AFL power schedules while handling timeouts and resets in long-running, stateful targets [15]–[21].

**Feedback signals.** Greybox coverage dominates in emulators (basic-block/edge coverage based on AFL++ QEMU/Unicorn) and on devices via coarse signals from debug/trace hardware [20], [22], [23]. Additional fitness includes memory-safety events and silent corruption checks, path length/depth, or execution time [24]–[28]. Coverage is imperfect but practical surrogate where ground-truth bugs are scarce [29], [30].

To reduce this saturation, recent fuzzing work adds *data-centric* feedback that tracks how values propagate, not just which blocks execute. Such signals (e.g., dependency- or data-flow-based coverage) can reward partial progress toward deep checks even when control-flow coverage does not change [8]–[10], [31].

**Bug detection/triage & throughput.** Firmware faults manifest as hard-faults, resets, or liveness loss rather than POSIX crashes; detection uses debug hooks, heartbeats, watchpoints, and sanitisers in emulation [32]–[38]. Emulators exploit parallelism and snapshot/restore; on device, watchdog loops reduce reflashing [22], [39], [40]. Methodologically, evaluations should emphasise diverse targets and bug-centric metrics, not coverage alone [20], [29]. Surveys underline that balancing fidelity, feedback quality, and speed is the central design tension [2], [41].

### B. Hardware fuzzing

We categorise hardware fuzzing by *where* execution occurs and *how* device interactions are realised: (1) on-device execution, (2) full/partial re-hosting in software.

*a) On-device (real hardware):* Firmware runs on the target MCU/SoC; inputs are injected via UART/USB/network/GPIO, and feedback is harvested through SWD/JTAG/CoreSight or trace units when available. Breakpoint-oriented approaches (µAFL, GDBFuzz) provide coarse coverage without binary instrumentation [20], [42]. Instruction-trace (Intel PT/ARM ETM) yields high-fidelity off-chip coverage at low perturbation when available [6], [43], [44]. Where debug/trace is locked down, side-channel signals (power/EM, timing) approximate path distinctions [45], [46]. Classical black-box interface fuzzing remains pragmatic for many IoT services, with stateful greybox variants improving effectiveness [47]–[50]. It can achieve perfect peripheral/timing fidelity and realistic concurrency. But the fuzzing will be limited with slower iteration, limited breakpoint/trace capacity, and scaling complexity (e.g., device pools, resets).

*b) Re-hosting (software emulation):* Firmware executes under QEMU/Unicorn built-in introspection and fast reset/snapshot, but requires sufficient peripheral realism to avoid dead-ends or false conclusions [51]–[55]. Automated peripheral modeling spans MMIO classification with fuzzer-supplied data ($P^2$IM), HAL call shims (HALucinator), and static-analysis-guided models (Fuzzware), significantly improving reachability over naive stubbing [56]–[60]. Hybrid HIT proxying (Avatar/Avatar$^2$) forwards selected MMIO to real hardware, combining emulator speed and coverage with high-fidelity I/O [5], [61]–[65]. Solver-assisted systems (Jetset,

$\mu$Emu) compute peripheral inputs to reach goals, reducing blind exploration [66], [67].

## III. Hardfuzz Overview

We propose an offline static analysis stage with an online fuzzing loop to systematically cover def-use chains on an embedded device. Figure 1 illustrates the overall architecture of Hardfuzz, which can be divided into three main phases: (1) Static Analysis & setup, (2) Def-use-guided fuzz loop, and (3) Coverage-driven input generation.

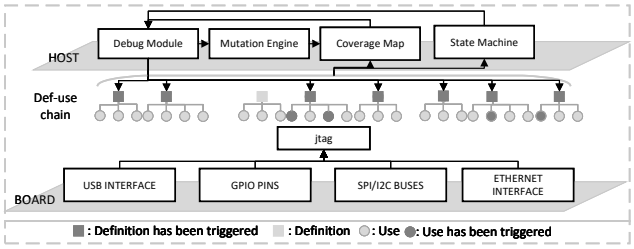


Fig. 1. Hardfuzz Overview

1) Static Analysis & Setup: Before fuzzing, we analyse the target program's binary to extract all def-use chains from reaching definition analysis. This yields a set of definition addresses each paired with one or more use addresses. The Hardfuzz runner loads this information and initialises its components: the GDB controller, serial connection, metrics logger, and input generator. The GDB controller attaches to the device or emulator and performs an initial reset/halt. The serial connection thread is started to handle input/output with the target. The input generator is seeded either with user-provided seed inputs or a default seed; it maintains the corpus of interesting inputs discovered. At this stage, Hardfuzz also precomputes some helper structures from the def-use list, such as a mapping of each def address to the basic block containing it (and likewise for uses). It also computes a weighting for each def (for fuzzing schedule) based on the number of uses it has.
2) Def-Use-Guided Fuzz Loop: Hardfuzz then enters the main fuzzing loop, which runs infinite rounds of test generation and execution. In each round, an input is selected and mutated, then used to execute a series of def-use chain trials. Unlike a pure coverage fuzzer that would run one input and simply note which new blocks were hit, Hardfuzz actively guides each input run towards a specific def-use target. It works as follows: it selects a subset of def addresses (up to the hardware breakpoint limit, e.g. 6) that have not yet been fully covered, and sets hardware breakpoints at those definition addresses (marked as Def-BPs). Then it releases the target to run the test input from the beginning. If none of those definitions execute (no breakpoint hit), the input did not trigger those targets; Hardfuzz will then try a different set of def addresses (or a new input in the next round). If one of the def breakpoints hits, the execution stops at that definition point. At this moment, Hardfuzz identifies which def was hit and retrieves its list of corresponding use addresses. It then immediately sets a second set of breakpoints for those uses (marking them Use-BPs) and resumes execution. The original def breakpoint, being temporary, is auto-removed upon hit to free a slot. Now the target continues running the same input, but with breakpoints set at the uses of the just-hit definition. If any of those uses executes, the program will halt again at the use site (indicating the def-use chain was successfully realised at runtime). Hardfuzz logs this as a def-use pair covered and removes the use breakpoint. It allows the program to continue, potentially catching multiple uses in one execution if the input triggers more than one use of the definition's value. Once the program completes (or a timeout/crash occurs), Hardfuzz cleans up any remaining breakpoints and resets the target if needed before the next round.
3) Coverage-Driven Input Generation: After each test execution, Hardfuzz updates its coverage bitmap to reflect any newly covered def or def-use pair. It uses two 64kB bitmaps in shared memory: one for def coverage (indexed by def address bits) and one for def-use coverage (indexed by a hash of def and use addresses). Any time a definition is hit, or a def-use pair is triggered, the corresponding bits are set. At the end of a round, Hardfuzz checks if any new bits were set compared to the global coverage map. If new coverage was found, the input that achieved it is saved to the corpus and considered for fuzzing again in the future. The fuzzer then chooses a new baseline input for mutation. It can choose the latest high-value input or cycle through the corpus to keep diversity. Hardfuzz employs a mutation engine based on libFuzzer's mutator [68]: by linking against the LLVM libFuzzer mutation library, it can generate mutated variants of an input efficiently. In each round, one or more new candidate inputs are produced this way. If a round produced no new coverage (no def-use hit and no new def hit), Hardfuzz can retry with a different def target or eventually switch to a fresh mutated input. This coverage-driven strategy ensures that Hardfuzz concentrates on inputs that expand the def-use coverage.

## IV. Def-Use Chain Analysis and Selection

We extract *definition-use (def-use) chains* from MCU's firmware binaries to guide test generation and breakpoint placement. A *definition site* (def) is an instruction that writes a program value (a register or a memory location). A *use site* (use) is an instruction that reads that value. A *def-use chain* is a directed line from a def instruction to a use instruction along some feasible path in the data dependence graph (DDG). We use these chains to (i) measure dataflow coverage and (ii) prioritise fuzzing inputs that reach definitions with many uses.

**Algorithm 1:** Def-Use Chain Extraction (per function)

**Input:** Function $F$, Data Dependence Graph $G$
**Output:** Set $\mathcal{E}$ of pairs $(\textit{def\_addr}, \textit{use\_addr})$

$\mathcal{E} \leftarrow \varnothing$
$\mathrm{RD} \leftarrow \textsc{ReachingDefinitions}(F)$
**foreach** $d \in \mathrm{RD}.\textit{all\_definitions}$ **do**
  $n_d \leftarrow \textsc{Node}(G, d.\textit{ins\_addr})$
  **if** $n_d = \perp$ **then continue**
  **foreach** $u \in \textsc{GetUses}(\mathrm{RD}, d)$ **do**
    $n_u \leftarrow \textsc{Node}(G, u.\textit{ins\_addr})$
    **if** $n_u = \perp$ **then continue**
    **if** $\textsc{Reachable}(G, n_d, n_u)$ **then**
      $\mathcal{E} \leftarrow \mathcal{E} \cup \{(d.\textit{ins\_addr},\ u.\textit{ins\_addr})\}$
**return** $\mathcal{E}$

Our analysis runs in three phases. First, we load the target ELF with `angr` and build a context-sensitive data dependence graph (DDG). For each discovered function, we run `ReachingDefinitions` based on angr's intermediate representation (VEX IR) to compute the set of definitions that may reach each program point. For each definition we found, we enumerate its uses with instruction address and check for reachability in the DDG and CFG. If there exists a path from the def to the use, we add an edge $\underbrace{a_d \to a_u}_{\text{address of def and use}}$ to the def-use graph. The graph also contains chains that cross function boundaries (e.g., def in caller, use in callee). The details are shown in Algorithm 1.

## V. Breakpoint Strategy

After extracting the def-use chains, we prioritise definitions to guide the fuzzer's exploration. The goal is to focus on definitions that influence many uses, as they are more likely to lead to diverse program behaviours and potential vulnerabilities. We also consider the history of selections to avoid over-focusing on a few definitions. We assign each definition a base weight equal to the minimum number of distinct uses it has, so definitions with many uses are considered more "interesting" by default. During fuzzing process, we adjust weights based on how often a def has been tried locally in the current round and globally across all rounds. Intuitively, if a particular def has already been hit several times (globally) or if we have attempted it repeatedly in the current round, its probability is reduced to avoid too much repetition. The exact formula is described below.

For a definition address $a_d$ with use set $U(a_d)$, the scheduler samples with

$$w(a_d) = \underbrace{\max\big(1,\ |U(a_d)|\big)}_{\text{base weight}} \cdot \underbrace{\frac{1}{1+\ell(a_d)}}_{\text{local penalty}} \cdot \underbrace{\frac{1}{\big(1+g(a_d)\big)^{1/2}}}_{\text{global penalty}}.$$

where $\ell(a_d)$ is the *local* count of selections of $a_d$ in the current generator and $g(a_d)$ is the *global* hit/selection count accumulated across rounds. Definitions are drawn by roulette-wheel sampling proportional to $w(a_d)$.

Once a def $a_d$ triggers, we order its uses by address proximity and enable up to $K$ hardware breakpoints (with $K = 6$ on ARM Cortex-M3):

$$\mathrm{order}_U(a_d) = \operatorname*{argsort}_{a_u \in U(a_d)} \big|a_u - a_d\big|, \tag{1}$$

$$S(a_d) = \text{first } K \text{ elements of } \mathrm{order}_U(a_d). \tag{2}$$

The intuition is that uses close to the def are more likely to be executed soon after the def, increasing the chance of hitting a use in the same run. If a def has more than $K$ uses, we will not be able to cover them all in one execution. However, since we sample defs multiple times across rounds, we will eventually cover all uses over time. Once one breakpoint hits, we will consider the basic block containing it as covered and remove the breakpoint to free a slot for the next use breakpoint. In this way, we can potentially catch multiple defs in one execution if the input triggers one basic block.

In each fuzzing round, hardfuzz draws up to K definition targets from this weighted generator (with K set to the hardware breakpoint limit) to form a batch. The reason for batching is efficiency: setting breakpoints is slow, and it is wasteful to run on input per breakpoint if we can enable multiple breakpoints at once. Batching also allows one input to potentially cover multiple defs if they happen to be hit in the same execution. The batch is constructed, and all breakpoints for that batch are inserted before running the test input. If none of breakpoints in the batch are hit by the time the input finishes, it implies the input does not execute any of those defs. In that case, Hardfuzz will fetch the next batch of defs (if any remain untried for this input) and rerun the same input on a fresh instance of the program. This approach gives each input multiple opportunities to demonstrate coverage on different def targets. If an input completely fails to hit any new def after exhausting all batches, Hardfuzz will conclude that the input is "stuck" coverage-wise and move to the next input. In our implementation, we set a limit (`e.g., NO_TRIGGER_THRESHOLD=8`) on consecutive attempts with no new hits before abandoning an input to avoid infinite loops.

The workflow of the breakpoint strategy is summarised in Algorithm 2. Managing the limited hardware breakpoints is a core part of Hardfuzz's design. We implement a lightweight GDB controller that communicates with the target device via GDB's machine interface (MI). The controller provides primitives to set and remove breakpoints, continue execution, wait for stops, and handle crashes or timeouts. These primitives are used in the breakpoint strategy to orchestrate the def-use guided execution.

When a def breakpoint triggers, the GDB stop reason comes as "breakpoint-hit" with an associated breakpoint number. We determine whether this was one of our def breakpoints by looking it up in the batch mapping. If so, we record the hit and prepare to switch to use breakpoints. Notably, on the Arm Cortex-M: when a breakpoint hits at an instruction in flash, the processor actually replaces that instruction with a

**Algorithm 2:** Hardware Breakpoint Strategy (Def→Use under comparator budget $K$)

**Input:** Test input $x$; batch `DefsBatch` $\subseteq \mathcal{D}$; use map $U(\cdot)$; HW breakpoint limit $K$
**Output:** `HitDef` $\in \mathcal{D} \cup \{\texttt{None}\}$
`HitPairs` $\subseteq \{(d, u)\}$

**foreach** $d \in$ *DefsBatch (up to $K$)* **do**
  SETHWBP($d$, `temporary` = `True`)
CONTINUEANDFEED($x$)
(`reason`, `payload`) ← WAITSTOP()
**if** *`reason` is "breakpoint hit" and payload is a def BP* **then**
  `HitDef` ← $d^\star$
**else if** *`reason`* $\in$ {*"timed out"*, *"crashed"*, *"exited'"*} **then**
  RESTARTIFCRASHEDORTIMEDOUT()
  **return** (`HitDef`, `HitPairs`)
HALTTHENDELETEALL()
**if** *`HitDef` = `None`* **then**
  **return** (`HitDef`, `HitPairs`)
`UsesSorted` ← uses in $U(d^\star)$ sorted by $|u - d^\star|$ (ascending)
**while** *untried uses remain* **do**
  take next chunk `UChunk` of $\leq K$ addresses from `UsesSorted`
  HALTTHENDELETEALL()
  **foreach** $u \in$ *`UChunk`* **do**
    SETHWBP($u$, `temporary` = `False`)
  CONTINUEANDFEED($x$)
  (`reason`, `payload`) ← WAITSTOP()
  **if** *`reason` is 'breakpoint hit" and payload is a use BP* **then**
    let $u^\star$ be the hit use
    `HitPairs` ← `HitPairs` $\cup \{(d^\star, u^\star)\}$
    REMOVEBP($u^\star$)
    **continue**
  **else if** *`reason`* $\in$ {*"timed out"*, *"crashed"*, *"exited"*} **then**
    RESTARTIFCRASHEDORTIMEDOUT(); **break**
  **else**
    **continue**
HALTTHENDELETEALL()
**return** (`HitDef`, `HitPairs`)

BKPT instruction internally. If we immediately removed the breakpoint and continued, we risk re-executing the BKPT instead of the original instruction. To avoid this, Hardfuzz performs a single-step operation to execute the instruction and move past it before inserting new breakpoints. This ensures the def instruction completes and the PC advances, preventing any "flash breakpoint deadlock" where the same breakpoint would re-trigger or corrupt execution. Our *BreakpointManager* handles this: upon detecting a def breakpoint number, it executes one instruction step, then clears all existing use breakpoints from any previous def, and finally removes the def breakpoint itself to free the slot. After that, Hardfuzz proceeds to install the use breakpoints for the triggered def.

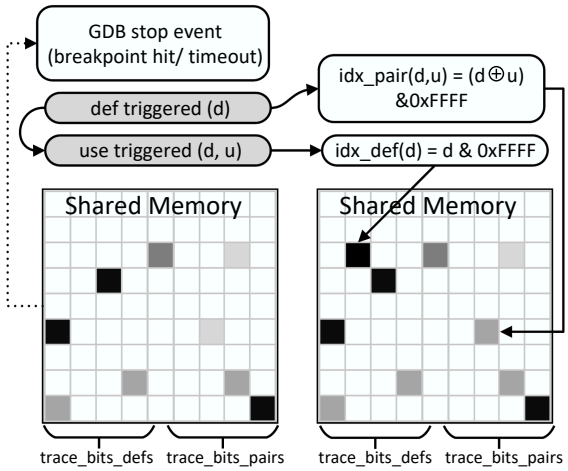


Fig. 2. Two-bitmaps in shared memory and update flow. Darkness indicates the time for the triggers to activate. bitmaps flip from $0x$FF to $0x00$ on first observation and gate corpus updates.

After each batch (or after a def-use sequence completes), Hardfuzz issues a blanket `-break-delete` command to clear any leftover breakpoints before moving on. This is important to prevent stray breakpoints from persisting into the next input's execution, which could cause false coverage signals or unintended halts. We found that after heavy churn of breakpoints, it was sometimes necessary to stabilise the GDB connection. In extreme cases (e.g., if the target becomes unresponsive or GDB misbehaves), Hardfuzz will restart the GDB session by killing the old GDB and launching a new one, then re-attaching to the target. This "GDB rejuvenation" is triggered after certain timeouts or errors to maintain a robust fuzzing run.

### A. Coverage Guidance

Hardfuzz needs a light-weight signal that can run on the device, without binary rewriting, and that still shows progress on data flow. We therefore record two events: (i) a definition is executed; and (ii) a definition-use pair is executed. We turn these events into coverage using two compact bitmaps stored in one shared-memory block (see Figure 2).

We allocate a single shared-memory region of size $2M$ bytes and split it into two non-overlapping slices: `trace_bits_defs`$[0{:}M-1]$ and `trace_bits_pairs`$[M{:}2M-1]$. In our implementation, $M = 65{,}536$. Each slice is a byte array used as a bitmap (0 or 1 per slot). This design lets the fuzzer and the coverage code communicate without copying and keeps the memory footprint fixed.

We map events to indices as follows: **Definition coverage,** when a def at address $d$ executes, `idx_def`$(d)$ $=$ $d$ & `0xFFFF`, and we set `trace_bits_defs`[`idx_def`$(d)$] $\leftarrow$ 1. **Def-use coverage,** when a use at address $u$ executes *after* the matching def at $d$ in the same input run, `idx_pair`$(d, u)$ $=$ $(d \oplus u)$ & `0xFFFF`, and we set `trace_bits_pairs`[`idx_pair`$(d, u)$] $\leftarrow 1$.

The XOR gives a constant-time hash from a pair of addresses to one slot. Collisions can happen but are rare at this scale; they may reduce granularity but do not break the guidance.

The figure illustrates three states of the shared-memory block: (i) **Initial**, where both bitmaps reflect the current round and no new hits have been recorded; (ii) **Def hit**, where one entry in `trace_bits_defs`$[0{:}M-1]$ flips to 1; and (iii) **Def–use hit**, where one entry in `trace_bits_pairs`$[M{:}2M-1]$ flips to 1.

Placing both slices inside one box with labels $[0, M)$ and $[M, 2M)$ emphasises that they share memory yet remain disjoint.

If the execution passes through a basic block without stopping inside it, we conservatively mark: (i) every def located in that block as covered; and (ii) every $(d, u)$ pair whose use lies in that block as covered. We do this using a precomputed lookup from each block to its defs and to the $(d, u)$ pairs whose $u$ is in that block. This avoids setting a breakpoint at every use site while still rewarding progress once the block executes.

To decide if an input should be kept, we maintain two bit arrays in process memory, `fresh_defs`$[0..M-1]$ and `fresh_pairs`$[0..M-1]$, initialized to $0x$FF. After running an input we scan the two shared bitmaps. For each index $k$:

- If `trace_bits_defs[k] != 0` and `fresh_defs[k] == 0xFF`, then set `fresh_defs[k] = 0x00`.
- If `trace_bits_pairs[k] != 0` and `fresh_pairs[k] == 0xFF`, then set `fresh_pairs[k] = 0x00`.

If at least one byte flips from $0x$FF to $0x00$, the input exposed new coverage. We then add the input to the corpus and optionally pick it (or a mutated child) as the next baseline. The shared bitmaps are cleared for the next input, while the virgin arrays keep the lifetime view of what has already been discovered.

Basic-block coverage rewards only new control flow. Our two-bitmap scheme adds a data-flow signal. The def bitmap rewards reaching a definition; the pair bitmap rewards reaching a use of that definition. These intermediate signals give the fuzzer a gradient toward the deep path even when no new basic block is covered.

## VI. Evaluation

### A. Experimental Setup

We evaluated Hardfuzz against GDBFuzz on two platforms: (1) an emulated environment using QEMU, and (2) a real hardware setup using an Arduino Due board (SAM3X8E MCU) connected via a J-Link debug probe connecting with host machine (x86-64 ubuntu 24.04 LTS). The fuzzing campaigns were run for a fixed time budget on each platform. For GDBFuzz, which does not natively track def-use chains, we consider only basic block coverage for comparison. All experiments used the same initial seed corpus and were allocated identical time for fairness.

### B. QEMU-Based Emulation Results

In the QEMU emulation, both fuzzers can execute inputs relatively quickly (no physical device latency). In this way, we could compare the two different breakpoint assignment strategies (Hardfuzz's def-use guided vs. GDBFuzz's Dominator-based) under the same conditions. We ran each fuzzer for 24 hours in this environment for three repetitions. The target programs we choose are from Google Fuzzbench [69], a well-known benchmark suite for fuzzing research. We selected 16 targets that are compatible with QEMU and also been tested in original GDBFuzz [20]. The results are shown in Figure 3.

According to figure 3, Hardfuzz outperforms GDBFuzz, achieving higher basic block coverage in 16 targets. This indicates that def-use chain guidance effectively directs the fuzzer towards more diverse and deeper code paths compared to the dominator-based strategy used by GDBFuzz. The results suggest that focusing on data flow relationships (def-use chains) provides a more informative signal for exploration than solely relying on control flow structures (dominators). In three cases, like freetype2, sqlite and lcms, GDBFuzz performs comparably or slightly better, which may be due to specific program structures where dominator-based selection happens to align well with critical paths. However, the overall trend favours Hardfuzz's approach, highlighting the benefits of incorporating data flow analysis into the fuzzing process. The gradients of the curves in the first few hours also suggest that Hardfuzz could discover new coverage faster initially, likely due to the richer feedback from def-use hits. This demonstrates that Hardfuzz's targeted approach succeeded in driving execution along rare data-flow paths that mere control-flow coverage did not prioritise.

The unique basic block results in bar Figure 4 further reinforce Hardfuzz's advantage. Over the 24-hour period, Hardfuzz consistently discovers more unique blocks than GDBFuzz, indicating that its def-use chain guidance effectively drives exploration into new areas of the codebase. The GDBFuzz can achieve similar results in only two targets (freetype2 and sqlite). This suggests that while dominator-based selection can be effective in certain scenarios, it generally lacks the nuanced direction provided by def-use analysis. The ability to target specific data-flow interactions allows Hardfuzz to uncover paths that may be overlooked when focusing solely on control-flow structures. The results highlight the importance of considering both control and data flow in fuzzing strategies to maximise coverage and discovery potential.

### C. On-Device Hardware Results

We evaluated Hardfuzz on a real device: an Arduino Due (SAM3X8E) connected through a J-Link. Running on physical

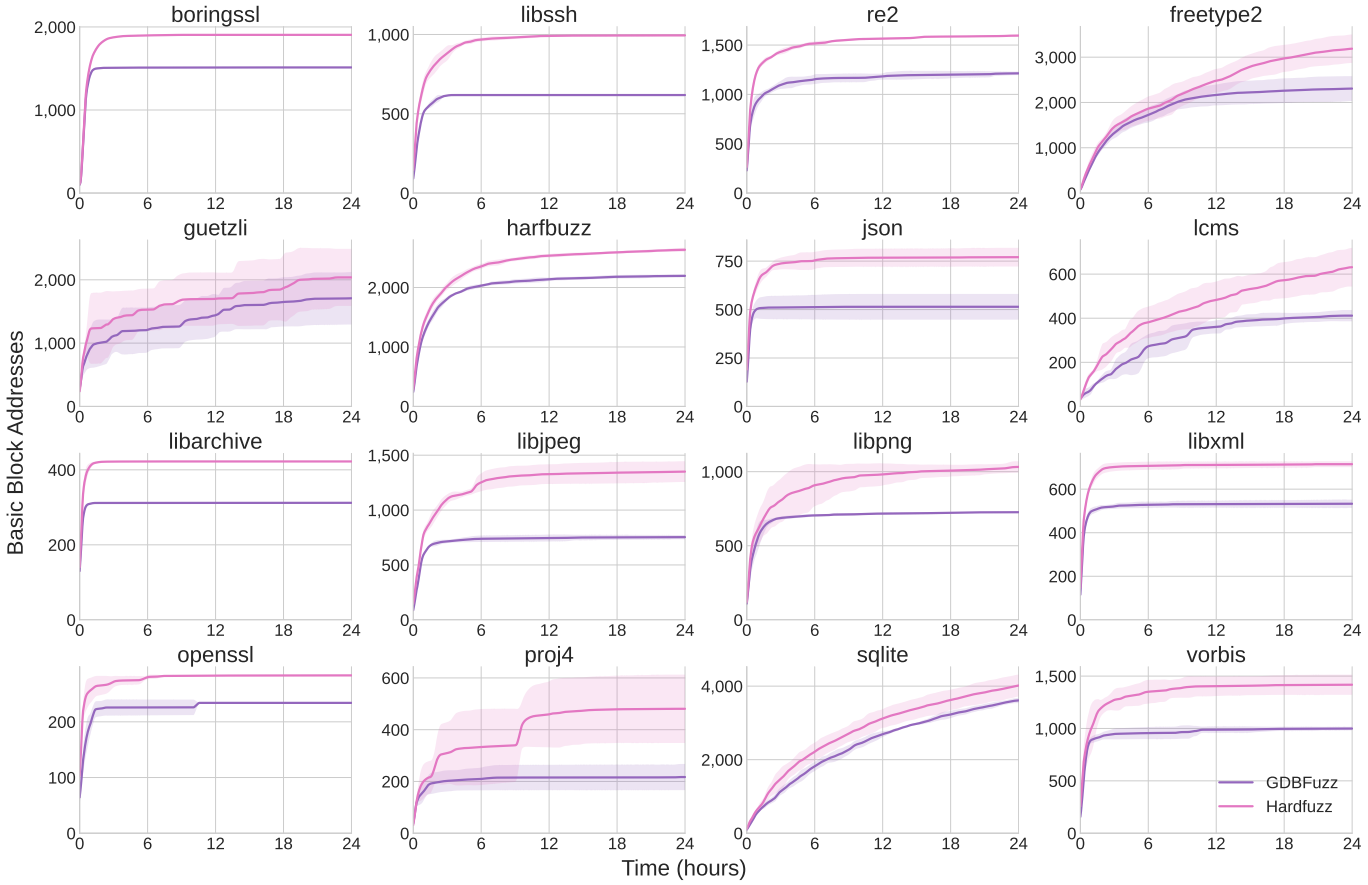


Fig. 3. QEMU Emulation Results: Basic block coverage achieved by Hardfuzz and GDBFuzz over 24 hours across 16 targets. Hardfuzz consistently outperforms GDBFuzz in most cases, demonstrating the effectiveness of def-use chain guidance in improving coverage.

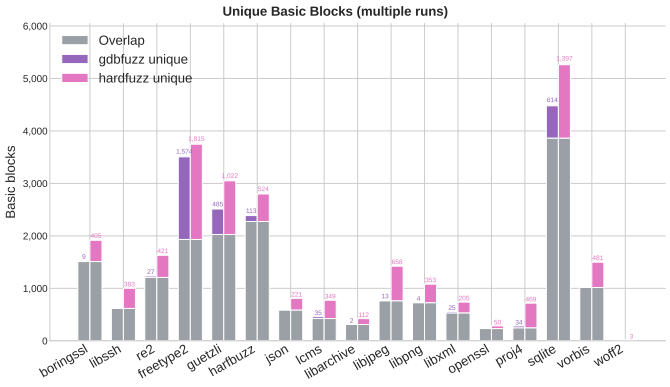


Fig. 4. Unique basic block coverage over time on QEMU. Hardfuzz consistently discovers more unique blocks than GDBFuzz, demonstrating its superior exploration capabilities.

hardware adds latency from the debug link and the lower clock speed of the MCU, but it gives us ground-truth signals (hardware faults and precise stop points). We ran both Hardfuzz and GDBFuzz for 24 hours with three repetitions on this setup. The firmware targets are the same types used in our GDBFuzz experiments, and each contains a small, known bug so we can measure detection and deduplication. The three targets are:

1) **buggycode (stack overflow).** A minimal UART harness that looks for the four-byte gate `"bug!"` and then copies the received payload into a fixed 20-byte stack buffer without bounds checks. Any input longer than 20 bytes triggers a deterministic overflow.
2) **HTTP server (state-machine bug).** A small ESP-IDF[1] HTTP service with an endpoint that mixes fixed-length responses with chunked sends in the same request. This violates the server's send path and produces a reproducible failure under load, modelling common handler mistakes in embedded web servers.
3) **JSON parser (length-triggered hang).** A serial JSON parser built with ArduinoJson that reads a 32-bit length prefix. If the length exceeds the configured buffer size, the firmware enters a persistent wait state. This gives us a clean timeout class distinct from crashes.

Table I summarises basic-block coverage on hardware after a 24-hour campaign. Across all three targets, Hardfuzz covers more blocks than GDBFuzz. These gains support our main claim: def-use guidance steers the fuzzer toward deeper and more diverse code paths than a dominator-based strategy. In

[1]https://github.com/espressif/esp-idf/tree/master

TABLE I
BASIC BLOCK COVERAGE ON HARDWARE AFTER 24 HOURS

| Target | Basic Blocks Covered | |
|---|---|---|
| | GDBFuzz | Hardfuzz |
| buggycode | 62/249 | 88/249 |
| HTTP server | 373/1504 | 524/1504 |
| JSON parser | 664/1071 | 758/1071 |

short, data-flow signals provide more informative guidance than control-flow structure alone, and the best results come from considering both.

Figure 5 shows coverage over time on hardware. Hardfuzz discovers new blocks faster in the early hours and maintains a lead throughout the run. This pattern matches the extra signals from def-use hits, even when a branch is not taken, observing a use of a value defined earlier helps retain and mutate promising seeds. For example, when a value had to be non-zero and also satisfy a range check before a sensitive operation, def-use guidance pushed mutations toward that combination earlier; GDBFuzz reached it later by chance.

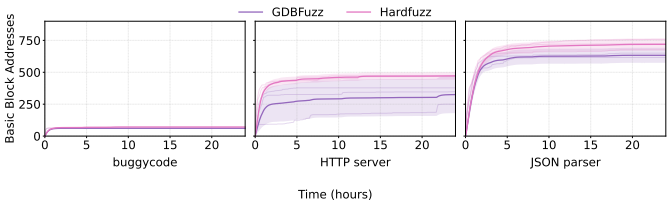


Fig. 5. Coverage changes over time on hardware.

Notably, Hardfuzz's higher coverage does not come from extra online analysis. We extract def-use chains with angr offline before fuzzing, so this step adds no runtime cost. By contrast, GDBFuzz updates control-flow information during fuzzing when it finds new coverage, which adds some overhead. Both systems rely on GDB stop reasons for crash detection; halts, breakpoint churn, and occasional re-attach process also cost time on real hardware. Despite these costs, Hardfuzz's richer signals yield better exploration and explain the observed coverage gains.

## VII. LIMITATIONS AND FUTURE DIRECTION

Hardfuzz improves fidelity and feedback on MCUs, but it does not replace emulation or rehosting. These approaches are complementary. Emulation scales and is easy to automate across many targets. On-device fuzzing gives ground truth behaviour but pays for I/O latency and debug overheads and is tied to specific boards.

Cortex-M parts expose only a small number of hardware comparators. Setting and reseting breakpoints through GDB adds latency, and some devices lack trace mechanisms entirely. This limits the number of def/use pairs we can watch at once and execution throughput. We plan to combine flash breakpoints with watchpoints, use RAM software breakpoints, amortise re-programming with persistent execution loops on the target, and opportunistically use trace (ETM/ITM/ETB) or RTT mailboxes when available to cut halt/resume cycles [70].

Semihosting, SWD/JTAG, and serial handshakes add delay. Our current design halts to set breakpoints and to step past breakpoints, which reduces cycles per second. Future work could design a persistent harness that processes many testcases per boot, a small on-target control loop to set next breakpoints via a memory-mapped index table without global halt.

We build def-use chains from binaries. Optimised builds, inlining and register allocation can obfuscate the mapping from IR to concrete addresses and drop some uses. Stripped binaries reduce function recovery quality. The future work includes: (1) fall back to dynamic analysis; (2) add lightweight dynamic taint or value-flow sampler to validate and refine static pairs; (3) consume optional symbols or minimal debug info when present; and (4) model common library idioms to cut false pairs.

As shown in Figure 3 and Figure 4, the code coverage growth eventually slows down and plateaus. This happens because both fuzzers use libfuzzer's mutation strategy, which relies on a random combination of simple changes like bit flips, byte flips, and arithmetic operations. While this method is effective for exploring a broad range of inputs initially, it often struggles to generate the specific inputs needed to bypass complex checks and reach deep program states. As a result, the fuzzer reaches a point of saturation, after which finding new code paths becomes rare, and coverage growth is negligible. Therefore, instead of running the fuzzer for a fixed, long duration like 24 hours, it is more practical to analyse this saturation point to determine an efficient time budget for the fuzzing campaign [71], [72].

On hardware we used three focused targets with known faults; in emulation we used a larger corpus. This is useful for controlled comparisons, but broader external validity needs more firmware, more boards, and blind bugs. Future work includes scaling to community firmware, report time-to-first-crash and deduped bug counts.

## VIII. CONCLUSION

This paper proposes Hardfuzz, an on-device dataflow-guided fuzzer for embedded systems. Hardfuzz uses hardware breakpoints to monitor def-use chains, providing precise and efficient feedback to guide the fuzzing process. The evaluation results show that Hardfuzz outperforms the state-of-the-art GDBFuzz in both emulated and real hardware environments, achieving higher code coverage and discovering more unique basic blocks. This demonstrates the effectiveness of def-use chain guidance in improving the exploration capabilities of fuzzers for embedded systems. Hardfuzz is available at https://github.com/MaksimFeng/Hardfuzz.

## ACKNOWLEDGMENTS

This work is supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant agreement EP/X037525/1 and the EU Horizon COCOON Project under grant agreement NO. 101120221.

## REFERENCES


[1] A. Mera, B. Feng, L. Lu, and E. Kirda, "DICE: Automatic emulation of DMA input channels for dynamic firmware analysis," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1938–1954.

[2] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Computing Surveys (CSUR)*, vol. 54, no. 1, pp. 1–36, 2021.

[3] N. Corteggiani, G. Camurati, and A. Francillon, "Inception:system-wide security testing of real-world embedded systems software," in *27th USENIX security symposium (USENIX security 18)*, 2018, pp. 309–326.

[4] K. Koscher, T. Kohno, and D. Molnar, "Surrogates: Enabling near-real-time dynamic analyses of embedded systems," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[5] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares." in *NDSS*, vol. 14, no. 2014, 2014, pp. 1–16.

[6] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 787–802.

[7] M. Zalewski, "American fuzzy lop," 2017, [Online]. Available: https://lcamtuf.coredump.cx/afl/.

[8] A. Mantovani, A. Fioraldi, and D. Balzarotti, "Fuzzing with data dependency information," in *2022 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2022, pp. 286–302.

[9] A. Herrera, M. Payer, and A. L. Hosking, "Dataflow: Toward a data-flow-guided fuzzer," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–31, 2023.

[10] M. Wang, J. Liang, C. Zhou, Z. Wu, J. Fu, Z. Su, Q. Liao, B. Gu, B. Wu, and Y. Jiang, "Data coverage for guided fuzzing," in *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 2024, pp. 2511–2526. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/wang-mingzhe

[11] S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Computing Surveys*, vol. 56, no. 3, pp. 1–38, 2023.

[12] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.

[13] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. De Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of DTLS implementations using protocol state fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2523–2540.

[14] J. Wang, L. Yu, and X. Luo, "Llmif: Augmented large language model for fuzzing iot devices," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 881–896.

[15] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "Morphuzz: Bending (input) space to fuzz virtual devices," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1221–1238.

[16] B. Yu, P. Wang, T. Yue, and Y. Tang, "Poster: Fuzzing iot firmware via multi-stage message generation," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 2525–2527.

[17] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, no. 191, 2022.

[18] P. Amini and A. Portnoy, "Sulley fuzzing framework," https://github.com/OpenRCE/sulley, 2010.

[19] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search." in *NDSS*, 2023.

[20] M. Eisele, D. Ebert, C. Huth, and A. Zeller, "Fuzzing embedded systems using debug interfaces," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1031–1042.

[21] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of RTL on FPGAs," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[22] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, 2020, pp. 10–10.

[23] J. Yun, F. Rustamov, J. Kim, and Y. Shin, "Fuzzing of embedded systems: A survey," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–33, 2022.

[24] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization." in *NDSS*, 2020.

[25] G. Zhang, P. Wang, T. Yue, X. Kong, S. Huang, X. Zhou, and K. Lu, "Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing," *arXiv preprint arXiv:2401.15956*, 2024.

[26] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, and L. Sun, "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*, 2022, pp. 417–428.

[27] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," in *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, 2021, pp. 337–350.

[28] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2541–2557.

[29] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.

[30] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "SoK: Prudent evaluation practices for fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1974–1993.

[31] T. E. Kim, J. Choi, K. Heo, and S. K. Cha, "DAFL: Directed grey-box fuzzing guided by data dependency," in *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, 2023, pp. 4931–4948. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/kim-tae-eun

[32] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, p. 6, 2018.

[33] M. Eceiza, J. L. Flores, and M. Iturbe, "Fuzzing the internet of things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems," *IEEE Internet of Things Journal*, vol. 8, no. 13, pp. 10 390–10 411, 2021.

[34] L. Seidel, D. C. Maier, and M. Muench, "Forming faster firmware fuzzers." in *USENIX Security Symposium*, 2023, pp. 2903–2920.

[35] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "Parmesan: Sanitizer-guided greybox fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2289–2306.

[36] Y. Jeon, W. Han, N. Burow, and M. Payer, "Fuzzan: Efficient sanitizer metadata design for fuzzing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 249–263.

[37] J. Schilling, A. Wendler, P. Görz, N. Bars, M. Schloegel, and T. Holz, "A binary-level thread sanitizer or why sanitizing on the binary level is hard," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1903–1920.

[38] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 754–768.

[39] Q. Liu, F. Toffalini, Y. Zhou, and M. Payer, "Videzzo: Dependency-aware virtual device fuzzing," in *2023 IEEE Symposium on security and privacy (SP)*. IEEE, 2023, pp. 3228–3245.

[40] E. Geretto, C. Giuffrida, H. Bos, and E. Van Der Kouwe, "Snappy: Efficient fuzzing with adaptive and mutable snapshots," in *Proceedings of the 38th annual computer security applications conference*, 2022, pp. 375–387.

[41] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory *et al.*, "Sok: Enabling security analyses of embedded systems via rehosting," in *Proceedings of the 2021 ACM Asia conference on computer and communications security*, 2021, pp. 687–701.

[42] W. Li, J. Shi, F. Li, J. Lin, W. Wang, and L. Guan, "$\mu$afl: non-intrusive feedback-driven fuzzing for microcontroller firmware," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1–12.

[43] T. Yue, F. Zhang, Z. Ning, P. Wang, X. Zhou, K. Lu, and L. Zhou, "Armor: Protecting software against hardware tracing techniques," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 4247–4262, 2024.

[44] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "Potus: Probing off-the-shelfusb drivers with symbolic fault injection," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[45] P. Sperl and K. Böttinger, "Side-channel aware fuzzing," in *European Symposium on Research in Computer Security*. Springer, 2019, pp. 259–278.

[46] P. Borkar, C. Chen, M. Rostami, N. Singh, R. Kande, A.-R. Sadeghi, C. Rebeiro, and J. Rajendran, "Whisperfuzz:white-box fuzzing for detecting and locating timing vulnerabilities in processors," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5377–5394.

[47] H. Liu, S. Gan, C. Zhang, Z. Gao, H. Zhang, X. Wang, and G. Gao, "Labrador: Response guided directed fuzzing for black-box iot devices," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1920–1938.

[48] Y. Zhang, W. Huo, K. Jian, J. Shi, H. Lu, L. Liu, C. Wang, D. Sun, C. Zhang, and B. Liu, "Srfuzzer: An automatic fuzzing framework for physical soho router devices to discover multi-type vulnerabilities," in *Proceedings of the 35th annual computer security applications conference*, 2019, pp. 544–556.

[49] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "Firmfuzz: Automated iot firmware introspection and analysis," in *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, 2019, pp. 15–21.

[50] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, no. 7, p. 191, 2022.

[51] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl:high-throughput greybox fuzzing of iot firmware via augmented process emulation," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1099–1114.

[52] K. Fang and G. Yan, "Emulation-instrumented fuzz testing of 4g/lte android mobile devices guided by reinforcement learning," in *European Symposium on Research in Computer Security*. Springer, 2018, pp. 20–40.

[53] F. Speiser, I. Szalay, and D. Fodor, "Embedded system simulation using renode," *Engineering Proceedings*, vol. 79, no. 1, p. 52, 2024.

[54] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[55] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices." in *NDSS*, 2018.

[56] B. Feng, A. Mera, and L. Lu, "P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1237–1254.

[57] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "Halucinator: Firmware re-hosting through abstraction layer emulation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1201–1218.

[58] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using precise mmio modeling for effective firmware fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1239–1256.

[59] D. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of emulation for kernelspace fuzzing," in *13th USENIX workshop on offensive technologies (WOOT 19)*, 2019.

[60] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae: Towards large-scale emulation of iot firmware for dynamic analysis," in *Proceedings of the 36th Annual Computer Security Applications Conference*, 2020, pp. 733–745.

[61] M. Eisele, M. Maugeri, R. Shriwas, C. Huth, and G. Bella, "Embedded fuzzing: a review of challenges, tools, and solutions," *Cybersecurity*, vol. 5, no. 1, p. 18, 2022.

[62] P. Cousineau and B. Lachine, "Enhancing boofuzz process monitoring for closed-source scada system fuzzing," in *2023 IEEE International Systems Conference (SysCon)*. IEEE, 2023, pp. 1–8.

[63] H. Peng and M. Payer, "Usbfuzz: A framework for fuzzing usb drivers by device emulation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2559–2575.

[64] H. Zhang, K. Lu, X. Zhou, Q. Yin, P. Wang, and T. Yue, "SIoTFuzzer: fuzzing web interface in iot firmware via stateful message generation," *Applied Sciences*, vol. 11, no. 7, p. 3120, 2021.

[65] D. Tychalas, H. Benkraouda, and M. Maniatakos, "ICSFuzz: Manipulating i/os and repurposing binary code to enable instrumented fuzzing in ics control applications," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2847–2862.

[66] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted firmware rehosting for embedded systems." in *USENIX Security Symposium*, 2021, pp. 321–338.

[67] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2007–2024.

[68] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 2016, pp. 157–157.

[69] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "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*, 2021, pp. 1393–1403.

[70] M. E. Weingarten, N. Hossle, and T. Roscoe, "High throughput hardware accelerated coresight trace decoding," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[71] N. Walkinshaw, M. Foster, J. M. Rojas, and R. M. Hierons, "Bounding random test set size with computational learning theory," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2538–2560, 2024.

[72] P. Tramontana, D. Amalfitano, N. Amatucci, A. Memon, and A. R. Fasolino, "Developing and evaluating objective termination criteria for random testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 3, pp. 1–52, 2019.