Characterizing and Bridging the Diagnostic Gap in eBPF Verifier Rejections

Yusheng Zheng*[1,2], Zhengjie Ji*[3], Weichen Tao*[2,4], Xiangyu Gao[5], Jianchang Su[6], Wei Zhang[6], Andi Quinn[1], Dan Williams[3]

[1]UC Santa Cruz [2]eunomia-bpf [3]Virginia Tech [4]Telecom Paris [5]University of Washington [6]University of Connecticut

Abstract

eBPF lets developers run custom programs inside the Linux kernel. The eBPF verifier simulates the program, tracking how the state of the registers and stack changes, and rejects any instruction it cannot prove safe. A rejection returns the states the verifier recorded while checking, and one verifier error message at the rejected instruction. To learn whether a rejection gives a developer enough information to guide a repair, we conduct an empirical study of 235 reproduced rejections. We find that the verifier error message does not identify the root cause mistake in the program's source: it maps to as many as nine distinct root causes. The recorded states hold only implicitly the safety property the rejected instruction required, so reaching a repair takes manual work. We present `bpfix`, a userspace tool that turns the recorded states into a Rust-style diagnostic: it states the proof the rejected instruction required and marks the source lines where that proof was established or lost. To test whether the diagnostic helps, we have three language models repair 75 rejected programs, judged by an oracle independent of `bpfix`. Every model repairs more programs with the `bpfix`-produced diagnostic than with the log, by 11 to 21 percentage points, and cuts token use by up to 71%. `bpfix` is available at https://github.com/eunomia-bpf/bpfix.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Empirical software validation**; *Software verification*; *Operating systems.*

ACM Reference Format:
Yusheng Zheng*[1,2], Zhengjie Ji*[3], Weichen Tao*[2,4], Xiangyu Gao[5], Jianchang Su[6], Wei Zhang[6], Andi Quinn[1], Dan Williams[3]. 2026. Characterizing and Bridging the Diagnostic Gap in eBPF Verifier Rejections. In *Proceedings of 4th Workshop on eBPF and Kernel Extensions (eBPF '26), September 29–October 2, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3837779.3838156

eBPF '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2911-9/26/09
https://doi.org/10.1145/3837779.3838156

1 Introduction

eBPF lets developers run custom programs inside the Linux kernel for networking, security, and observability [1, 4]. A developer compiles the program to bytecode and submits it to the kernel, which loads the program only after the verifier proves it safe [5, 11]. The verifier simulates the program, tracking the type and range of the value in each register and stack slot [11]. It rejects an instruction when a safety property that instruction requires, referred to as a *proof*, does not hold. The verifier stops at that instruction and returns a log of the recorded states and one verifier error message.

To successfully load the program, the developer has to find and fix the cause of the rejection, but the log does not identify it. Figure 1 shows one such rejection from Stack Overflow. The program bounds-checks a packet pointer against `data_end` (Figure 1(a), line 267), yet the verifier rejects a later load through that pointer with `R5 invalid mem access 'scalar'` (Figure 1(b)). The log shows the value already as a scalar before the load, but the verifier error message reports only the load. Finding where the value stopped being considered a pointer and started being considered a scalar means reading a trace written in registers and instruction indices and mapping it back to source by hand.

We call the distance between what a rejection returns and what a repair needs the *diagnostic gap*. To measure it, we reproduce 235 rejections from public reports and kernel selftests and label each with its *root cause*, the source-level error the developer made that ultimately led to rejection (§2). We find that the verifier error message does not determine the root cause: the most frequent message covers nine distinct causes across 28 rejections. The recorded states do hold what the message omits, but only implicitly, so recovering a root cause from them is manual work. Many of the root causes are specific to the eBPF verifier, so that work also takes knowledge of the verifier's own rules.

While BPF debugging tools exist, no existing tool recovers the root cause from a rejection. Pretty Verifier [14] maps

*Equal contribution.

the verifier's error messages to C source lines using compiler debug information, and the BPF Verifier Visualizer [8] renders the recorded states in a debugger-like interface, but neither derives the root cause. Outside of BPF, type-error diagnosis expresses the checker analysis as constraints and, when the constraints conflict, ranks the program expressions most likely to be the cause [13, 22]. The eBPF verifier builds no such structure; the log records a sequence of states, and the onus is on the developer to reconstruct the root cause.

To bridge the diagnostic gap, we present `bpfix`, a userspace command-line tool that works from the verifier's log alone. Our insight is that the recorded states already hold the proof the rejected instruction needed, so `bpfix` can recover it without re-analyzing the program. `bpfix` identifies that proof at the rejection site, then tracks the proof backwards through the earlier states. The output is a Rust-style diagnostic that states which proof the rejected instruction required and marks the source lines where that proof was established or lost.

We evaluate `bpfix` on `bpfix-bench`, a benchmark of 75 rejected programs, by giving its diagnostic output to language models and testing the fixes they produce. Each of three models sees every program twice, once with the raw verifier log and once with the `bpfix` diagnostic. Each model-proposed fix is successful if it loads and passes the case's tests. We find that every model repairs more programs with `bpfix`'s diagnostic: from the raw verifier log the models repair 0–37% of the programs in one shot, and the diagnostic adds 11–21 percentage points. On Qwen3.6 27B the diagnostic also cuts token use by up to 71%.

This paper makes the following contributions:

- An empirical study of 235 reproduced BPF verifier rejections, showing that what a rejection returns does not determine the cause a repair has to address (§2).
- `bpfix`, a userspace command-line tool that recovers the root cause from the verifier's log and reports it as a source-level diagnostic (§3).
- `bpfix-bench`, a benchmark of 75 rejected programs on which the `bpfix` diagnostic improves repair for three language models (§5).

(a) C Code

```
263 if (ipv4_hdr)
264     udph = (void *)ipv4_hdr + sizeof(*ipv4_hdr);
265 else
266     udph = (void *)ipv6_hdr + sizeof(*ipv6_hdr);
267 if (udph + sizeof(struct udphdr) > data_end)  // bounds-checked
268     return 1;
269
270 dst_port = __constant_ntohs(((struct udphdr *)udph)->dest); // rejected
```

(b) Verifier Log

```
from 31 to 34: ... R5_w=40 ...
; if (udph + sizeof(struct udphdr) > data_end) @ prog.c:267
35: (07) r3 += 8                        ; R3=48
36: (2d) if r3 > r2 goto pc+4           ; R2=pkt_end() R3=48
; dst_port = ...((struct udphdr *)udph)->dest; @ prog.c:270
37: (69) r2 = *(u16 *)(r5 +2)
R5 invalid mem access 'scalar'
```

(c) `bpfix` output

```
error[BPFIX-E006]: verifier-visible compiler lowering hides the required
       proof
  = class: lowering_artifact
  = confidence: medium
  = next action: provenance
  --> prog.c:270
   |
263 | if (ipv4_hdr)
    | ------------- nearby source context for pointer provenance
267 | if (udph + sizeof(struct udphdr) > data_end)
    | ------------------------------------------- verifier state changes
          from pkt to scalar before the rejected access
270 | dst_port = __constant_ntohs(((struct udphdr *)udph)->dest);
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ rejected
          here: verifier sees a scalar where a pointer is required
   |
   = verifier[229]: R5 invalid mem access 'scalar'
   = required proof: preserve a verifier-recognized pointer type at the
         operation that requires a pointer
help: Reacquire a verifier-tracked pointer before the rejected dereference.
help: Use the packet pointer that received the data_end proof, or rederive
       and recheck it before the load.
```

Figure 1. A rejection from Stack Overflow 53136145. The source bounds-checks `udph` at line 267 (a), yet the verifier rejects the load at line 270, where the register holds a scalar (b). From the same log, `bpfix` states the proof the load required and maps the relevant states back to source (c).

2 An Empirical Study of Verifier Rejections

This section tests whether a rejection report from the BPF verifier identifies the *root cause*, or the source-level error the developer made that resulted in rejection. We conduct an empirical study of 235 real reproduced BPF verifier rejections. We determine the categories of what actually causes each rejection, and whether the verifier error message adequately identifies that cause.

2.1 Background: The Verifier Rejection

Before the kernel runs an eBPF program, the verifier must prove the program safe by simulating one instruction at a time over every path, recording for each register and stack slot the *proof* that its value is used safely (e.g., the byte range a bounds check gives a packet pointer). A program that fails a check is rejected, and the verifier returns a log with two parts: the per-instruction *recorded states* it kept while checking, and then a single line accompanying the rejected instruction that we refer to as the *verifier error message*.

For example, in Figure 1(b), the rejected load at instruction 37 reads R5, which fails a proof (only pointers, not scalars, can be dereferenced). However, the log already shows R5 is a scalar on entry to instruction 34. Thus the root cause occurred earlier, when R5 became a scalar. More generally, the rejection location accompanying the verifier error message is not always where the repair must land. To fix the issue the developer must reconstruct, by tracing back through the program, how and where the offending state originated and how it propagated through the program.

```c
// correct source; -O0 lowers the field read into a scalar
static int add_one(int x) { return x + 1; }
int prog(struct xdp_md *ctx) { return add_one(ctx->rx_queue_index) & 1; }
```

```
verifier: R2 invalid mem access 'scalar'
```

Figure 2. A compiler artifact: correct source that the verifier reports as `invalid mem access 'scalar'`.

Table 1. Where the repair lands for the 235 rejections. `bpfix` reports the same four categories for a new rejection (§3.3).

Category	The repair changes	Cases
source-level bug	the program's source	191
compiler artifact	a compiler flag, not the source	18
environment	the kernel or program type	14
verifier	the program's structure	12
Total		235

2.2 Reproduction of Real Verifier Rejections

To study rejections that developers actually hit, we assemble `bpfix-empirical`, a dataset of real verifier rejections. The dataset draws on 936 candidates from four sources: Stack Overflow questions, GitHub issues, GitHub fix commits, and kernel selftests. We rebuild each candidate and load it into kernel 6.15.11 with clang 18 at log level 2. 235 of the 936 candidates reject under this build and form `bpfix-empirical`. We drop the rest because they do not reject under our toolchain, need a specific environment, or lack the source to rebuild.

Each `bpfix-empirical` case carries the faulty source and the developer's own fix. The source and the fix give the ground truth for the rejection and where its repair landed, which the next two subsections use.

2.3 Characterization of Verifier Rejections

We characterize each `bpfix-empirical` rejection by where its repair lands, in the four categories of Table 1. In 191 of the 235 rejections the repair changes the program's source; we call these *source-level bugs*. The other 44 are rejections of correct source; for example, Figure 2 shows a *compiler artifact*: `-O0` lowers a provably safe context-field read into a scalar, and a compiler flag repairs the rejection without any source change. Other rejections of correct-source programs involve the kernel or BPF environment (e.g., the program's type does not allow a helper it calls), or the verifier (e.g., the program exceeds the verifier's stack limit).

We group each of the 191 source-level bugs into 12 root cause categories, shown in Table 2, by analyzing and clustering their repairs. Note that the root causes are generally not deficiencies that all programming languages will prohibit or warn against, though some checks, like null checks and object capacity checks, are done by many. In general, addressing a root cause requires an understanding of the eBPF verifier's rules.

Table 2. The 191 source-level bugs and the 12 root-cause categories they fall into.

Root-cause category	Cases
Unclamped scalar used as offset or length	24
Corrupted or stale dynptr object	23
Packet access without a bound on every path	22
Missing null check	19
Pointer type or provenance mismatch	16
Unverified address dereferenced	16
Index exceeds object capacity	15
Context or contract misuse	15
Unpaired resource reference	15
Interrupt flag not restored in order	11
Probe signature mismatched with the ABI	9
Oversized or uninitialized stack buffer	6
Total	191

Takeaway #1: Most rejections (81%) are source-level bugs that require knowledge of the verifier's rules to fix.

2.4 Measurement of the Diagnostic Gap

To learn how well the verifier error message identifies the root cause, we mask the registers and numeric literals in each message and group the 235 rejections by normalized template. The 235 rejections carry 167 distinct messages, which fall into 82 templates. We find that 15 of the 82 templates appear in more than one root cause. Table 3 lists the four most frequent verifier error message templates. The most frequent template spans nine root causes across 28 rejections.

Table 3. The four most frequent verifier error message templates, with the cases sharing each template and the distinct root-cause categories they span. A template normalizes registers and offsets to placeholders.

Verifier error message template	Cases	Categories
`R# invalid mem access 'scalar'`	28	9
`invalid access to packet`	26	5
`invalid access to map value`	18	4
`R# !read_ok`	13	4

Takeaway #2: The verifier error message does not determine the root cause, since one message can match as many as nine distinct causes.

3 `bpfix` Design and Implementation

`bpfix` operates on the verifier log after the kernel verifier rejects a program, as shown in Figure 3. `bpfix` answers two questions the verifier error message leaves open: which proof

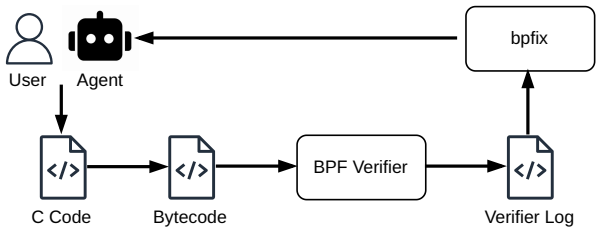

Figure 3. Where `bpfix` runs. A developer or an agent compiles the program to bytecode and submits it to the kernel; when the verifier rejects the program, `bpfix` reads the log the verifier returned and produces a diagnostic. `bpfix` changes neither the compiler nor the kernel.

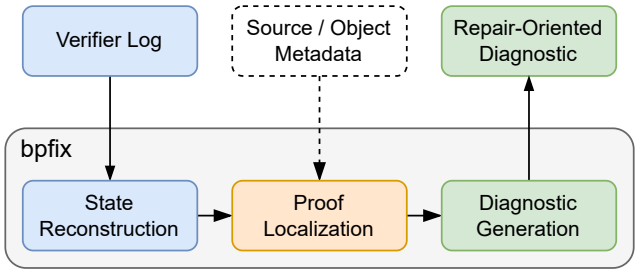

Figure 4. Overview of the `bpfix` workflow. `bpfix` analyzes verifier logs, reconstructs the missing proof, and generates a repair-oriented diagnostic.

the verifier required, and where that proof is missing. It then uses the answers to these questions to identify the root cause and produce a helpful error message for the developer. `bpfix` is implemented in approximately 23k lines of Rust and works in the three stages of Figure 4.

3.1 State Reconstruction

To locate where a proof was established or lost, the first stage reconstructs the state (information about values in registers and stack slots) the verifier held at each instruction. The existing log (under `log_level=2`) only provides the full state information at a few points, such as where the verifier resumes a pending branch. Elsewhere the log shows only what an instruction used.

`bpfix` therefore builds each state by accumulation, beginning from an instruction with full state, applying the per-instruction changes that follow, and starting over at the next full state or at a change of stack frame. The log also carries a source line for many instructions, and `bpfix` keeps that line with the instruction, so that the later stages can report on source lines rather than on register numbers.

3.2 Proof Localization

The verifier error message does not reveal which proof the rejected instruction required, so the second stage recovers that proof. Proofs that assert the same kind of fact form a *proof family*. For example, every packet-bounds proof says that an access stays inside a checked range, whatever the register or offset. `bpfix` distinguishes 18 families (e.g., packet-bounds for packet pointer range checks, scalar-range for general value constraints) and matches each verifier error message to one of them.

The family gives the kind of proof, and the recorded states around the rejection identify the specific proof instance. In Figure 1, `R5 invalid mem access 'scalar'` matches the pointer-provenance family, and the recorded states show that the rejected load reads `R5`, which therefore had to still be a packet pointer.

Once the required proof is identified, `bpfix` localizes where it was lost by following that proof back through the earlier recorded states. In the running example, an earlier state holds `R5` as `pkt(off=34,r=42)` and a later one holds it as the scalar `40`, so `bpfix` reports instruction 37 as the *rejection location* and identifies where the proof was lost. When no earlier state holds the proof, `bpfix` reports it as never established.

3.3 Diagnostic Generation

The last stage produces the diagnostic and chooses what to recommend. The diagnostic gives the rejected instruction, the proof it required, the source lines where that proof was established or lost, and which category of Table 1 the rejection falls in.

Within each family, `bpfix` defines finer-grained patterns, each combining the verifier error message, the rejected instruction, and the recorded states to identify one specific reason the required proof does not hold. For example, one pattern matches a 32-bit register copy that materializes a packet pointer as a scalar. Each pattern carries its own recommendation. A rejection that matches no pattern takes its recommendation from its proof family instead. Finally, the loss appears in the log as a change in register state, and in the diagnostic as the source lines a repair would change.

4 Case Studies

`bpfix` reports the proof a rejected instruction required and the source lines behind it, and whether that report matches the repair a developer has to make is a separate question. This section examines two of the 235 rejections whose repairs are on public record, so each diagnostic can be checked against the fix that was actually made. The first rejection shows what the diagnostic adds to a verifier error message, and the second shows it separating two rejections that one message alone cannot tell apart.

4.1 Recovering a Missing Proof

Figure 5 shows a rejection from the aya project. The program takes the address of the map object `&globals`, casts it to

(a) Rejected program

```
__u64 *raw_map = (__u64 *)&globals; // &globals: the map-object pointer
__u64 v = *raw_map;
*raw_map = v + 1;                   // rejected here
```

(b) Verifier error message

```
verifier: only read from bpf_array is supported
```

(c) bpfix diagnostic

```
error[BPFIX-E010]: map pointer is accessed as ordinary memory
  = class: source_bug              --> prog.c:20
  = required proof: derive a map-value pointer with the proper
    map helper before reading or writing map contents
help: Look up an element, then write through that pointer.
```

(d) The developer's fix

```
__u32 key = 0;
__u64 *v = bpf_map_lookup_elem(&globals, &key);
if (!v) return 0;
*v += 1;
```

Figure 5. A real rejection from the aya project (issue 1002): the rejected program (a), the verifier error message (b), the bpfix diagnostic (c), and the developer's fix (d).

__u64 *, and writes through the result. The verifier rejects the write with only read from bpf_array is supported. The message is accurate, and it names the operation the verifier refused, a write through the map object where a map value is required, without saying which pointer the write should have used.

bpfix reports the proof the write required, a map-value pointer derived through a map helper before any access to map contents, and recommends looking up an element and writing through the returned pointer. The aya developers repaired the program without that diagnostic, and their fix (Figure 5(d)) is the one it recommends: they call bpf_map_lookup_elem, check the returned pointer, and write through the result.

4.2 Separating a Source-Level Bug from a Compiler Artifact

Figure 6 shows two rejections that carry the same verifier error message, invalid mem access 'scalar', and belong to two different categories of Table 1. In Figure 6(a) the program builds a packet pointer from an integer offset, (void *)(sizeof(*eth) + nh_off), so the value never carries a packet base, and the later field read uses a scalar where the verifier requires a pointer. bpfix reports a source-level bug, which the diagnostic prints as source_bug.

In Figure 6(b) the source establishes the packet bound it needs, deriving udph from an IPv4 or IPv6 header and checking it against data_end before the read. The rejection appears only after the compiler merges the two branches, and the merged pointer no longer carries the type the verifier tracks. bpfix reports a compiler artifact, printed as lowering_artifact, and the repair upstream confirms it: the developers changed how the program is compiled and left the source logic alone.

(a) Source-level bug (Stack Overflow 56965789)

```
if (eth->h_proto == bpf_htons(ETH_P_IP)) {
    struct iphdr *iph2 = (void *)(sizeof(*eth) + nh_off); // integer offset,
          no packet base
    return iph2->protocol;                                  // rejected here
}
```

```
error[BPFIX-E011]: scalar or pkt_end value is used where the verifier
      requires a real pointer
  = class: source_bug
  = verifier[31]: R1 invalid mem access 'scalar'
```

(b) Compiler artifact (Stack Overflow 53136145)

```
if (ipv4_hdr)
    udph = (void *)ipv4_hdr + sizeof(*ipv4_hdr);
else
    udph = (void *)ipv6_hdr + sizeof(*ipv6_hdr);
if (udph + sizeof(struct udphdr) > data_end)  // udph (merged) bounds-checked
    return 1;

dst_port = __constant_ntohs(((struct udphdr *)udph)->dest); // rejected here
```

```
error[BPFIX-E006]: verifier-visible compiler lowering hides the required
      proof
  = class: lowering_artifact
  = verifier[229]: R5 invalid mem access 'scalar'
```

Figure 6. Two rejections that share the verifier error message invalid mem access 'scalar' and fall in different categories of Table 1: a source-level bug (a) and a compiler artifact (b). Each diagnostic shows only its error id, class, and the verifier error message it explains.

A developer who reads invalid mem access 'scalar' cannot tell from it whether to change the program or the build. bpfix answers that question in both rejections, and in the second the upstream repair confirms the answer.

5 Evaluation

The evaluation asks three research questions about effectiveness across models, token efficiency, and component contribution. We answer them with bpfix-bench, a benchmark of 75 rejected programs for language models to repair.

5.1 Benchmark Construction

bpfix-bench contains 75 repair tasks: 40 constructed to exercise specific required proofs and 35 minimized from open-source projects such as Cilium [1], xdp-tools [17], and bpftime [25]. Each task includes the rejected program and an executable oracle independent of bpfix. The oracle first compiles and loads each fix. It then applies case-specific preservation checks to the accepted verifier log, the candidate source, or both. These checks cover proofs and source operations that the program's outputs and map updates do not capture. Finally, functional tests execute the repaired program and check those outputs and map updates. The oracle accepts the fix only if every step passes.

We test two attempt budgets. One-shot mode allows one model call, while retry mode allows one failure-informed second call. Our primary model is Qwen3.6 27B, and we also run the hosted GLM 5.2 and the smaller Qwen2.5 3B at temperature zero. The 3B model provides a lower-capacity

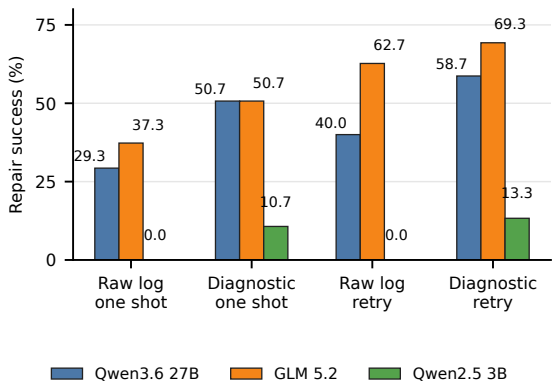

Figure 7. bpfix-bench repair success across Qwen3.6 27B, GLM 5.2, and Qwen2.5 3B. Bars compare the raw verifier log with the bpfix diagnostic in one-shot and retry mode.

comparison. For each task, a model receives the rejected program with either the raw verifier log or the bpfix diagnostic and writes a candidate fix for the oracle to judge.

5.2 RQ1: How Effective Is bpfix Across Models?

Figure 7 shows that every model repairs more tasks with the bpfix diagnostic than with the raw verifier log. Qwen3.6 27B and GLM 5.2 both show gains with the diagnostic under both attempt budgets. The small Qwen2.5 3B repairs none of the 75 tasks from the raw verifier log, but repairs 8 with the diagnostic in one shot.

We assign each unsuccessful one-shot repair to its first observed failure: model call, compilation, verifier load, preservation check, or functional test. Table 4 shows that the diagnostic mainly reduces failures at verifier load and at the preservation check across all models.

With the raw verifier log, 62 one-shot candidates from the small Qwen2.5 3B fail at verifier load. Three prompts containing the raw verifier log exceed the model's context window and return no program. With the shorter bpfix diagnostic, model-call failures drop to zero and verifier-load failures to 39, while compilation failures rise from 7 to 14.

Answer to RQ1: Across all three models, the diagnostic increases one-shot repair success by 11–21 percentage points and reduces failures at verifier load.

5.3 RQ2: How Token Efficient Is bpfix?

A separate Qwen3.6 27B run measures repair success and token use on the same 75 tasks. Its raw verifier log repair count differs slightly from RQ1, so every comparison below stays within this run (Table 5(a)).

The model server, which has a 32K-token context window, reports token use for 72 of 75 one-shot calls with the raw verifier log and 122 of 129 calls with the raw verifier log under the two-attempt budget. It rejects the other three and seven calls before returning a usage record. Every token count for the raw verifier log in Table 5(a) is therefore a lower bound. Usage records are complete for bpfix under both attempt budgets. Caching and the fixed run order prevent a fair wall-time comparison, so we compare only tokens.

Table 5(a) shows that bpfix reduces total token use by at least 70.7% in one shot and at least 67.7% with one retry, while repairing more tasks under both budgets.

Answer to RQ2: In this Qwen3.6 27B run on these 75 tasks, bpfix repairs more tasks while using fewer tokens under both attempt budgets.

Table 4. First observed failure for one-shot repairs on bpfix-bench, in evaluation order: model call, compilation, verifier load, preservation check, and functional test. Preservation checks are case-specific checks of the accepted verifier log, the candidate source, or both. Input is either the raw verifier log (raw) or the bpfix diagnostic (bpfix). In the header, Preserve denotes the preservation check and Func. denotes the functional test. For each row, successful repairs plus these failures sum to 75.

Model	Input	Call	Compile	Load	Preserve	Func.
Qwen3.6 27B	raw	0	3	19	22	9
	bpfix	0	1	10	16	10
GLM 5.2	raw	0	1	10	25	11
	bpfix	0	1	5	22	9
Qwen2.5 3B	raw	3	7	62	3	0
	bpfix	0	14	39	8	6

5.4 RQ3: What Contributes to bpfix's Effectiveness?

We run a separate one-shot ablation on Qwen3.6 27B using four inputs. Source only contains the rejected program. Raw verifier log tail adds the largest suffix of complete lines from the output of each task's final verifier load that fits the per-task input-token limit. bpfix without recommendations removes every line beginning with help: from the diagnostic. The Full bpfix diagnostic retains all diagnostic lines. We report Source only as a descriptive reference. The first paired comparison asks whether bpfix without recommendations helps beyond raw verifier log tail under the same per-task input-token limit. The second asks whether the recommendation lines included in the Full bpfix diagnostic, prefixed help:, improve repair. For each task, the raw verifier log tail uses no more input tokens than bpfix without recommendations. Every condition runs all 75 tasks once with the same Qwen3.6 27B model using Q4_K_M (4-bit) quantization, temperature zero, a 32K-token context window, an 8,192-token output limit, and reasoning disabled. This ablation is separate from RQ1 and the token measurement in RQ2.

Table 5. Repair success and token use for Qwen3.6 27B across 75 tasks. Panel (a) compares the raw verifier log with the `bpfix` diagnostic under two attempt budgets; panel (b) reports a separate one-shot ablation. Input tokens include cached tokens. Total tokens equal input tokens plus output tokens. The ≥ symbol marks lower bounds for calls with the raw verifier log that lack usage records.

(a) Token comparison

Attempt budget	Input	Repairs	Input tokens	Cached tokens	Output tokens	Total tokens	Tokens per repair
One shot	Raw log	21/75	≥359,072	≥1,512	≥40,506	≥399,578	≥19,028
One shot	`bpfix`	38/75	73,684	1,575	43,475	117,159	3,083
Up to one retry	Raw log	32/75	≥702,022	≥60,028	≥75,734	≥777,756	≥24,305
Up to one retry	`bpfix`	44/75	185,314	2,352	66,002	251,316	5,712

(b) One-shot ablation

Input	Repairs	Input tokens	Cached tokens	Output tokens	Total tokens	Tokens per repair
Source only	24/75	50,627	1,575	51,829	102,456	4,269
Raw log tail	26/75	68,274	1,575	51,451	119,725	4,605
`bpfix` without recommendations	29/75	70,206	1,575	43,768	113,974	3,930
Full `bpfix` diagnostic	38/75	73,534	1,554	51,769	125,303	3,297

Table 5(b) reports the results. The raw verifier log tails use 68,274 input tokens, or 97.25% of the 70,206 tokens used by `bpfix` without recommendations. We run two paired comparisons and report 95% confidence intervals.

The first paired comparison remains uncertain on these 75 tasks. `bpfix` without recommendations repairs 29 tasks, compared with 26 for the raw verifier log tail, a difference of 4.0 percentage points (95% CI, −4.2 to 12.1 percentage points). Six tasks succeed only with `bpfix` without recommendations, while three succeed only with the raw verifier log tail (exact $p = 0.5078$; Holm-adjusted $p = 0.5078$). The confidence interval includes zero. `bpfix` without recommendations also uses 4.8% fewer total tokens and reduces tokens per successful repair from 4,605 to 3,930.

The recommendation lines improve repair on these tasks. The Full `bpfix` diagnostic repairs 38 tasks, compared with 29 without recommendations, a difference of 12.0 percentage points (95% CI, 3.3 to 20.2 percentage points). Ten tasks succeed only with the Full `bpfix` diagnostic, and one only without recommendations (exact $p = 0.0117$; Holm-adjusted $p = 0.0234$). The full diagnostic uses 9.9% more total tokens than the diagnostic without help lines, and it cuts tokens per successful repair by 16.1%, from 3,930 to 3,297.

Answer to RQ3: The recommendation lines add nine repairs and reduce tokens per successful repair by 16.1%. `bpfix` without recommendations repairs three more tasks than the raw verifier log tail, but the confidence interval includes zero.

6 Related Work

eBPF verification and safety. Prior eBPF tooling decides whether a program is safe or works around the verifier, and leaves the developer to diagnose a rejection [3]. Offline reprovers such as PREVAIL establish safety over their own abstract domain [5], verifier audits check the soundness of its operators [16, 18, 19], and other work replaces the verifier with safe Rust [6], synthesizes passing programs [24], or isolates programs at runtime [9, 10, 12, 23]. None of them identifies which proof the program failed to establish.

Error diagnosis and fault localization. Outside eBPF, diagnosis reads a structure the checker keeps: type-error diagnosis reasons over typing constraints [13, 15, 22], spectrum-based fault localization over many test runs [7, 21], program slicing over the dependence graph [20], and unsat-core extraction over a solver's search [2]. A rejection returns only a text log, so none of these structures is available. For eBPF, Pretty Verifier maps the verifier error message to a source line and a hint with regular expressions [14], and BPF Verifier Visualizer renders the recorded states in an interactive frontend [8], but neither reports where the proof was lost. `bpfix` recovers that proof from the recorded states alone.

7 Conclusion

To measure the diagnostic gap between what a rejection returns and what a repair needs, we conduct an empirical study of 235 reproduced eBPF verifier rejections. The verifier error message does not identify the root cause, and the recorded states hold the missing proof only implicitly, so reaching a repair takes manual work and knowledge of the verifier's rules. We present `bpfix`, a userspace tool that recovers from the log alone what a repair needs: the proof the rejected instruction required, and the source lines that had to establish it. Each of the three language models repairs 11 to 21 percentage points more of the 75 rejected programs with that diagnostic than with the log itself.

References

[1] Cilium Authors. 2026. Cilium: eBPF-based Networking, Observability, and Security. https://github.com/cilium/cilium. Open-source project, accessed 2026.

[2] A. Cimatti, A. Griggio, and R. Sebastiani. 2011. Computing Small Unsatisfiable Cores in Satisfiability Modulo Theories. *Journal of Artificial Intelligence Research* 40 (April 2011), 701–728. doi:10.1613/jair.3196

[3] Mugdha Deokar, Jingyang Men, Lucas Castanheira, Ayush Bhardwaj, and Theophilus A. Benson. 2024. An Empirical Study on the Challenges of eBPF Application Development. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions* (Sydney, NSW, Australia) *(eBPF '24)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3672197.3673429

[4] Bolaji Gbadamosi, Luigi Leonardi, Tobias Pulls, Toke Høiland-Jørgensen, Simone Ferlin-Reiter, Simo Sorce, and Anna Brunström. 2024. The eBPF Runtime in the Linux Kernel. arXiv:2410.00026 [cs.OS] https://arxiv.org/abs/2410.00026

[5] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1069–1084. doi:10.1145/3314221.3314590

[6] Jinghao Jia, Ruowen Qin, Milo Craun, Egor Lukiyanov, Ayush Bansal, Minh Phan, Michael V. Le, Hubertus Franke, Hani Jamjoom, Tianyin Xu, and Dan Williams. 2025. Rex: Closing the language-verifier gap with safe and usable kernel extensions. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, Boston, MA, 325–342. https://www.usenix.org/conference/atc25/presentation/jia

[7] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (Long Beach, CA, USA) *(ASE '05)*. Association for Computing Machinery, New York, NY, USA, 273–282. doi:10.1145/1101908.1101949

[8] libbpf. 2025. bpfvv: BPF Verifier Visualizer. https://github.com/libbpf/bpfvv. Accessed 2026-06-25.

[9] Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. 2023. Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions* (New York, NY, USA) *(eBPF '23)*. Association for Computing Machinery, New York, NY, USA, 42–48. doi:10.1145/3609021.3609301

[10] Soo Yee Lim, Tanya Prasad, Xueyuan Han, and Thomas Pasquier. 2024. SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions. In *Proceedings of the 2024 on Cloud Computing Security Workshop* (Salt Lake City, UT, USA) *(CCSW '24)*. Association for Computing Machinery, New York, NY, USA, 80–94. doi:10.1145/3689938.3694781

[11] Linux kernel documentation. 2025. eBPF verifier. https://docs.kernel.org/bpf/verifier.html. The Linux Kernel documentation, Documentation/bpf/verifier.rst.

[12] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. 2024. MOAT: Towards Safe BPF Kernel Extension. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 1153–1170. https://www.usenix.org/conference/usenixsecurity24/presentation/lu-hongyi

[13] Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding minimum type error sources. *SIGPLAN Not.* 49, 10 (Oct. 2014), 525–542. doi:10.1145/2714064.2660230

[14] Rosario Rizza, Riccardo Sisto, and Fulvio Valenza. 2025. Design and implementation of a tool to improve error reporting for eBPF code. In *2025 IEEE International Conference on Cyber Security and Resilience (CSR)* (Chania, Crete, Greece). IEEE, Piscataway, NJ, USA, 214–219. doi:10.1109/CSR64739.2025.11130075

[15] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to blame: localizing novice type errors with data-driven diagnosis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 60 (Oct. 2017), 27 pages. doi:10.1145/3138818

[16] Hao Sun and Zhendong Su. 2024. Validating the eBPF verifier via state embedding. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 33, 14 pages.

[17] The XDP Project. 2026. xdp-tools: Utilities and Example Programs for Use with XDP. https://github.com/xdp-project/xdp-tools. Open-source project, accessed 2026.

[18] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2022. Sound, precise, and fast abstract interpretation with tristate numbers. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '22)*. IEEE Press, Piscataway, NJ, USA, 254–265. doi:10.1109/CGO53902.2022.9741267

[19] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. Verifying the Verifier: eBPF Range Analysis Verification. In *Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part III* (Paris, France). Springer-Verlag, Berlin, Heidelberg, 226–251. doi:10.1007/978-3-031-37709-9_12

[20] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, California, USA) *(ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449.

[21] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization . *IEEE Transactions on Software Engineering* 42, 08 (Aug. 2016), 707–740. doi:10.1109/TSE.2016.2521368

[22] Danfeng Zhang and Andrew C. Myers. 2014. Toward general diagnosis of static errors. *SIGPLAN Not.* 49, 1 (Jan. 2014), 569–581. doi:10.1145/2578855.2535870

[23] Peihua Zhang, Chenggang Wu, Xiangyu Meng, Yinqian Zhang, Mingfan Peng, Shiyang Zhang, Bing Hu, Mengyao Xie, Yuanming Lai, Yan Kang, and Zhe Wang. 2024. HIVE: A Hardware-assisted Isolated Execution Environment for eBPF on AArch64. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 163–180. https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-peihua

[24] Yusheng Zheng, Yiwei Yang, Maolin Chen, and Andrew Quinn. 2024. Kgent: Kernel Extensions Large Language Model Agent. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions* (Sydney, NSW, Australia) *(eBPF '24)*. Association for Computing Machinery, New York, NY, USA, 30–36. doi:10.1145/3672197.3673434

[25] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, Xiaozheng Lai, Dan Williams, and Andi Quinn. 2025. Extending applications safely and efficiently. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation* (Boston, MA, USA) *(OSDI '25)*. USENIX Association, Boston, MA, USA, 557–574. https://www.usenix.org/conference/osdi25/presentation/zheng-yusheng