

PatchAudit: Uncovering Faulty Security Patches and Lingering Zero-Days at Scale

Zhongsong Zhang

University at Buffalo, SUNY
Buffalo, NY, United States
zhengson@buffalo.edu

Haipeng Cai*

University at Buffalo, SUNY
Buffalo, NY, United States
haipengc@buffalo.edu

Abstract

Security patches are intended to eliminate vulnerabilities, but the initial fixes themselves can be flawed—leaving systems vulnerable even after patching. We present the first large-scale measurement of the underexamined class of *manifestly faulty patches*—initial CVE fixes that are later amended by follow-up commits, often silently. To enable this study, we developed PatchAudit, a novel, automated system that detects such faulty patches at scale by combining LLM-guided reasoning over the patching intent of code changes with static program analysis, cross-validated through a multi-pronged verifier. Using PatchAudit, we analyze 6,535 CVEs across 1,649 GitHub repositories over 26 years, and identify and manually validate 617 initial patches (9.4%) that were incomplete or incorrect.

We find that incomplete patches are more common and take longer to converge, often requiring multiple revisions to cover overlooked inputs, interfaces, or code paths. Incorrect patches, while less prevalent, tend to cause deeper regressions. We construct the first structured taxonomy of why patches fail, revealing that patches for different classes of vulnerabilities break in distinct ways—e.g., access control patches often miss alternate entry points, while input validation fixes struggle with input variation. Critically, PatchAudit uncovers 13 faulty patches that remain unfixed in current main branches; 7 have been confirmed by developers as real, exploitable zero-days. Our findings expose persistent structural blind spots in vulnerability remediation and show how automated patch auditing can enable actionable, ecosystem-wide insights into patch quality and lingering security risk.

CCS Concepts

• Security and privacy → Software and application security.

Keywords

Manifestly faulty, bad patch, incomplete patch, incorrect patch

ACM Reference Format:

Zhongsong Zhang and Haipeng Cai. 2026. PatchAudit: Uncovering Faulty Security Patches and Lingering Zero-Days at Scale. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26)*, November 15–19, 2026, The Hague, Netherlands. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3830454.3846785>

*Haipeng Cai is the corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License. CCS '26, The Hague, Netherlands

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
<https://doi.org/10.1145/3830454.3846785>

1 Introduction

The security of software systems is critical as cyber threats continually evolve and impose substantial risks on organizations and individuals [3, 5, 7]. Public statistics from the NVD [20] indicate the pace and scale of disclosure—48,244 CVE records published in 2025. A vital defense against these vast-growing threats is timely patching of the security vulnerabilities.

Yet speed alone is not enough—ensuring patch quality is essential. Security patches must completely and correctly restore the intended security property without introducing regressions or new flaws [13, 32]. In practice, however, a patch may be *faulty*: it either (a) fails to fully eliminate the targeted vulnerability [10, 13, 26, 31] (i.e., an *incomplete* fix) or (b) introduces a new security defect (e.g., authorization gaps, logic errors) [18, 29, 30, 34] during remediation (i.e., an *incorrect* fix). Such a patch can leave the system exploitable and, worse, create a false sense of security that delays or even impedes time-sensitive remediation.

Studies on patch quality exist (e.g., [13, 26]), using the presence of *future commits* [13] (or *supplementary patches* [26]) relative to the initial patch commit as the indicator of that patch being unreliable. However, they relied heavily on manual audit of commit logs to identify the presence. The most recent study [31] examines supplementary patches using structural heuristics such as location or diff similarity, which fail to capture semantic fix intent and miss most real future fixes [26]. As a result, existing understanding of faulty patches is confined to the limited scope, low coverage, and small scale of the underlying studies. Moreover, deeper characteristics of patch failures (e.g., recurring failure-mode patterns, severity of defects introduced by incorrect patches, and the real-world exploitability of the exposure window between a faulty patch and its correction) remain largely unknown.

To fill these gaps, we conduct a large-scale measurement study of security patch quality—focusing on *manifestly faulty patches* whose deficiencies are later addressed by follow-up commits of the same vulnerability-remediation intent, often without explicit disclosure. Our overarching insight is that (1) these manifestly faulty patches can be detected as per the presence of any *true* future commit and, more importantly, (2) a true future commit can be identified based on the semantic intent of its code changes to continue addressing the same vulnerability as the patch under audit. Conducting such a study at scale requires fully automated detection that goes beyond explicit references and reasons about code-level patching intent—a capability that no existing tool provides. We therefore developed PatchAudit, a fully automated pipeline for this purpose.

PatchAudit, featuring an *agentic and auditable architecture* that combines LLM-driven reasoning with static code analysis in an orchestrated workflow, advances current research by addressing several key challenges in effective patch auditing. The **first** is *intent*

disambiguation even with explicit evidence: even when a commit mentions a CVE ID or prior patch hash, the reference may not indicate that the commit continues the same security fix—reverts, cherry-picks, merges, and multi-issue commits can cite CVEs for bookkeeping rather than continuation. To address this challenge, PatchAudit uses an agentic LLM-based design to interpret commit messages and CVE mentions using graded evidence, determining whether the commit operates in the same vulnerability context and pursues the same security property.

The **second** challenge lies in recognizing *semantic continuity when no textual anchors are present*. Many corrective commits lack any explicit reference to prior patches, and diff-only comparisons fail when code shifts location or refactors. To address this challenge, PatchAudit identifies the code region where both the initial patch and the candidate future commit semantically interact via a lightweight static analysis. Within this narrowed, shared scope, a task-specific LLM agent evaluates whether the candidate commit continues or corrects the same security intent.

However, *LLMs, while capable of code reasoning, suffer from hallucinations, context overflow, and limited reliability when applied naively*. PatchAudit addresses this **third** challenge by building task-specific LLM agents and incorporating a verification module that comprises an independent LLM judge and an ensemble of static analyzers, which jointly evaluate the core agents' reasoning and assess semantic alignment with vulnerability states.

These careful design choices allow us to detect faulty patches both with and without textual anchors and to audit historical patch quality at our scale. Our evaluation confirms the effectiveness and efficiency of PatchAudit in detecting and classifying faulty patches. Using PatchAudit to analyze commit histories and patch flows across 6,535 CVEs in 1,649 GitHub repositories from 1999 to 2025, we identify and manually validate 617 manifestly faulty initial patches—9.4% of all analyzed CVEs. Notably, only 35.8% of these manifestly faulty patches had follow-up commits with explicit references; the rest was detected only via semantic intent analysis—confirming that reference-based approaches (used by existing studies) would miss the majority.

Our study reveals that manifestly faulty security patches are neither rare nor random, but a measurable and recurring phenomenon shaped by consistent structural patterns. Rather than isolated mistakes, these fixes reflect systemic weaknesses in patch design and verification. They disproportionately affect certain vulnerability classes, often persist undetected in production, and in many cases, create or preserve real exploitability. Among others, we found:

- **Incomplete patches dominate**: 74.1% of manifestly faulty patches are incomplete; they converge more slowly (median 114.6 vs. 67.4 days) and require more follow-ups than incorrect patches.
- **Incorrect patches are dangerous**: Among 160 manifestly incorrect patches, 88.8% introduced substantive security-relevant defects, and 43.1% resulted in directly exploitable flaws (e.g., use-after-free, authentication bypass, memory corruption).
- **Two-thirds of manifestly faulty patches are under-scoped**: 63.4% of manifestly faulty patches apply the "right" fix *conceptually*, but fail to reach the full exploitable surface—e.g., missing specific/boundary inputs, interfaces, or call paths. The remaining 36.6% stem from flawed realization, where the implemented logic is incorrect or unstable.

- **Patch failures follow structural patterns**: A two-dimensional taxonomy of patch failure *causes* and *resurfaced issues* reveals dominant pairings tied to vulnerability type—e.g., propagation drift in access control; undercoverage in input validation.
- **Silent failures are rising**: After year 2020, 77% of manifestly faulty patches lacked CVE or patch references in their corrective commits—up from 48% pre-2020—making reference-based faulty patch detection approaches increasingly ineffective.
- **Manifestly faulty patches enable real exploitation**: In 12 cases, public exploits or security advisories emerged during the exposure window between the manifestly faulty patch and its eventual fix, indicating their real-world security impact, not merely potential risks.
- **Manifestly faulty patches accumulate greater maintenance burden**: Compared to good security patches, manifestly faulty patches more often require multiple rounds of fixes and expanded coverage across the codebase, especially in the incomplete class.
- **Zero-day manifestly faulty patches remain live**: We identify 13 manifestly faulty CVE patches that remain unfixed in latest main-branch snapshots across 13 different projects, often due to interface inconsistencies or missed instances of the same vulnerable logic; these lingering zero-days have been validated by working PoCs and reported to respective developers via responsible disclosure; 7 have been confirmed by the developers and 1 assigned a new CVE so far.

In summary, the key contributions of this work include:

- **New technique and tool**: We developed PatchAudit, a novel, automated analysis framework for effectively discovering and characterizing manifestly faulty patches at scale (§3, §5).
- **New dataset**: We curated a sizable set of 617 manifestly faulty CVE patches (457 incomplete, 160 incorrect) spanning 239 projects, released to support future research on security patch quality (§4).
- **New understanding**: We ran PatchAudit on public CVEs with GitHub-tracked fixes across 26 years, and revealed the prevalence, severity, and failure-mode patterns of manifestly faulty patches, addressing the knowledge gap about the current problematic and consequential vulnerability remediation practice, informing how to improve it for various stakeholders (§6, §7).

2 Background and Motivation

We define key concepts and motivate our approach on patch quality.

2.1 Faulty Security Patches

A security patch is faulty if it is incomplete—it tries to address a vulnerability but leaves some aspect of that defect still present [13], or incorrect—it successfully removes the original vulnerability but inadvertently introduces a new security defect.

We focus on manifestly faulty patches—faulty patches with follow-up fixes, as these provide *concrete evidence* of patch failure. This evidence-based scope ensures reliable ground truth and enables scalable, verifiable measurement—an important first step in tackling faulty patches in general. Meanwhile, it yields a *lower bound* on prevalence because deficient patches that are never revisited fall outside our study.

The difference between the two faulty-patch types can be seen through a buffer-overflow example. The original code copies an input string into a fixed-size buffer without checking its length:

```

1 void process_input(char *input) {
2   char buffer[10];
3   strcpy(buffer, input); // Vulnerable: no bounds check
4   printf("Processed:%s\n", buffer); }

```

An incomplete patch may introduce a length check but mishandle boundary cases. The fix below rejects overly long inputs but permits inputs of length exactly 10, leaving no room for the null terminator:

```

1 - strcpy(buffer, input); // Vulnerable to buffer overflow
2 + if (strlen(input) > 10) { printf("Input_too_long!\n"); return; }
3 + strcpy(buffer, input); // Still vulnerable when strlen(input) == 10

```

By contrast, an incorrect patch may eliminate the overflow but introduce a new defect: e.g., replacing the unsafe copy with `snprintf` but using attacker-controlled input as the format string:

```

1 - strcpy(buffer, input); // Buffer overflow vulnerability
2 + snprintf(buffer, sizeof(buffer), input); // Format-string info leak

```

To reason about when two commits are addressing the same vulnerability, we use the notion of (security) *patching intent*, defined as the triplet (context, property, implementation). The context identifies the concrete vulnerability being addressed; the property denotes the security invariant being re-established (such as bounds safety, input validation, or authorization); and the implementation describes how that property is enforced. Evidence that two commits share the same intent may come from textual signals (e.g., references to a CVE ID or patch hash) or semantic signals in the code (e.g., diffs that enforce the same property along corresponding paths).

Then, we define a future commit for a CVE patch as any later commit that shares its patching intent. Thus, backporting, refactoring, and other security-irrelevant commits do not qualify as future commits. So, a patch can be detected as *manifestly faulty* (bad for short hereafter) by the presence of any true future commit for it. If the latest future commit, although intending to, still does not successfully resolve the bad patch and eliminate the vulnerability, that bad patch becomes a lingering, zero-day bad patch.

Also, we consider patch quality from a security-impact perspective: patches that *only* break functionality are thus not in scope.

Note that faulty fixes differ from general software flaws as the former occur inside the vulnerability-remediation pipeline—once the initial patch C_1 is labeled as the CVE fix, advisories, package managers, vendors, and downstream users may treat the vulnerability as closed, delaying effective patching. If C_1 is incomplete or incorrect, the system enters a misleading state of “patched but still exposed”, and downstream backports may apply only C_1 while missing silent corrective commits. This motivates studying patch chains, intent continuity, exposure windows, and advisory metadata—dimensions not captured by ordinary vulnerability or flawed-code studies.

2.2 Motivating Examples

Bad patches often surface in subtle ways: some later commits explicitly acknowledge that an earlier security fix was incomplete, while others silently repair the same deficiency without naming the original CVE. We illustrate both evidence scenarios below.

Explicit Reference. In PHP (CVE-2014-4049), shown in Figure 1, the initial patch attempted to harden DNS TXT-record parsing in `ext/standard/dns.c`, but only added a local length check and still left out-of-bounds reads possible. A later commit then explicitly stated that the earlier CVE fix was incomplete and strengthened the parser with systematic boundary checks.

```

Commit message: Fix potential segfault in dns_get_record()
Commit ID: b34d7849ed90ced9345f8ealc59bc8d101c18468

```

File Path: ext/standard/dns.c

```

1 while (ll < dlen) {
2   n = cp[ll];
3 + if ((ll + n) >= dlen) {
4 +   n = dlen - (ll + 1); }
5   memcpy(tp + ll, cp + ll + 1, n);

```

Commit message: Incomplete fix for CVE-2014-4049

Commit ID: 2fefae47716d501aec41c1102f3fd4531f070b05

File Path: ext/standard/dns.c

```

1 + static u_char *php_parserr(u_char *cp, u_char *end, ...)
2 + #define CHECKCP(n) do {if (cp+(n)>end) return NULL;} while (0)
3 ...
4 + CHECKCP(1);
5   n = cp[0];
6 ...
7 + CHECKCP(n);
8   memcpy(tp + ll, cp + 1, n);

```

Figure 1: Bad patch (top commit) with explicit future commit.

```

Commit message: Properly store certificate exceptions in Java viewer
Commit ID: f029745f

```

File Path: java/com/tigervnc/rfb/CSecurityTLS.java

```

1 + File vncDir = new File(FileUtils.getVncHomeDir());
2 + if (!vncDir.exists())
3 +   throw new AuthFailureException();
4 + File dbPath = new File(vncDir, "x509_known_hosts");

```

Commit message: Create VNC homedir if it doesn't already exist

Commit ID: 89f6d5ae

File Path: java/com/tigervnc/rfb/CSecurityTLS.java

```

1 if (!vncDir.exists()) {
2 -   throw new AuthFailureException("Could not obtain VNC home...")
3 + try {
4 +   vncDir.mkdir();
5 + } catch (SecurityException e) {
6 +   throw new AuthFailureException("Could not obtain VNC home...")

```

Figure 2: Bad patch with *implicit* future commit.

No Explicit Reference. In TigerVNC (CVE-2020-26117), shown in Figure 2, the initial patch corrected certificate-exception handling but introduced a new failure mode: if the VNC home directory did not already exist, the viewer threw an exception immediately and blocked legitimate first-time use. A later commit fixed this behavior without mentioning the CVE or initial patch reference, instead changing the same function to create the directory on demand.

Despite this difference in evidence, both cases require the same underlying judgment: determining whether a later commit continues the same security objective as an earlier patch—captured through our notion of *patching intent*. Focusing on this intent rather than mere references provides a unified way to recover both explicit and implicit future commits, and hence bad patches.

3 Design of PatchAudit

We first give an overview and then describe each component (phase).

3.1 Overview

The overall design is depicted in Figure 3. PatchAudit takes the (typically initial) patch commit C_1 to audit as its **Input**, retrieves all post- C_1 commits, and iterates through each (C_2) to analyze whether it continues C_1 ’s patching intent.

In Phase 1, applied first to every candidate, PatchAudit extracts C_2 ’s potential patching intent by scanning its commit log for any reference serving as *explicit* evidence (e.g., a CVE id or the hash of C_1). When such evidence is found, a dedicated LLM validates whether the reference implies C_2 is a future commit of C_1 , yielding a decision with a confidence score; a high-confidence decision is routed directly to Phase 3 for verification. If the extraction fails or the confidence falls below a threshold, PatchAudit instead proceeds to Phase 2 for implicit intent analysis: it intersects the forward

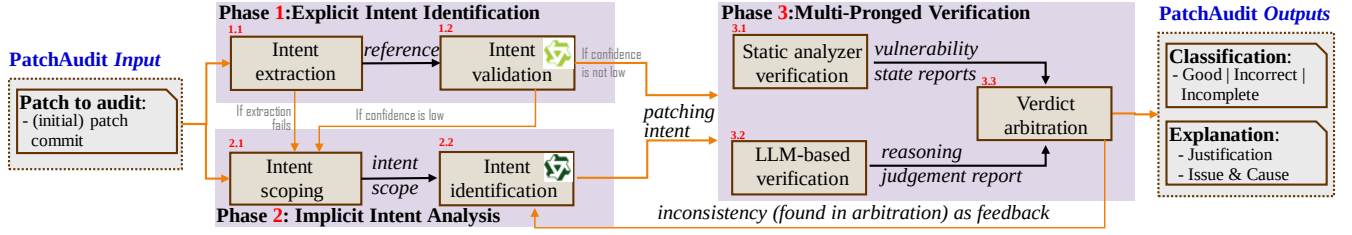


Figure 3: Overview of PatchAudit, including its input, three main working phases, and outputs.

Algorithm 1: Overall workflow of PatchAudit

```

Input:  $C_1$  (initial patch commit)
Output:  $label \in \{\text{Good}, \text{Bad}\}, \text{BadType} \in \{\text{Incomplete}, \text{Incorrect}\},$ 
 $explanation = (\text{justification}, [\text{issue}, \text{cause}])$ 

1 Function DetectBadPatch( $C_1$ )
2    $candidates \leftarrow \text{GetLaterCommits}(C_1)$  // Get later commits after  $C_1$ 
3   foreach  $C_2 \in candidates$  do
4      $Reference \leftarrow \emptyset, Confidence \leftarrow 0, Intent \leftarrow \emptyset$ 
5     // Phase 1: Explicit Intent Identification
6      $(status, Reference) \leftarrow \text{PotentialIntentExtraction}(C_1, C_2)$ 
7     if  $status = \text{Success}$  then
8        $(Confidence, Intent) \leftarrow \text{IntentValidation}(C_1, C_2, Reference)$ 
9     if  $Intent = \emptyset$  or  $Confidence$  is low then
10      // Phase 2: Intent Scoping
11       $InScope \leftarrow \text{IntentScoping}(C_1, C_2)$  if  $InScope = \emptyset$  then
12        continue // no shared code region; not future commit
13      // Phase 2: Implicit Intent Analysis
14       $(Confidence, Intent) \leftarrow \text{IntentIdentification}(C_1, C_2, InScope)$ 
15       $iter \leftarrow 0$ 
16      while  $iter < \text{max\_iters}$  do
17        // Phase 3: Multi-Pronged Verification
18         $(verdict, feedback) \leftarrow \text{MultiProngedVerification}(C_1, C_2, Intent, Confidence)$ 
19        if  $verdict = \text{future-commit}$  then
20           $(label, \text{BadType}, explanation) \leftarrow \text{ClassifyAndExplain}(C_1, C_2)$ 
21          return  $(label, \text{BadType}, explanation)$ 
22        if  $verdict = \text{not-future-commit}$  then
23          break
24        if  $verdict = \text{unresolved}$  then
25          // Phase 2: Re-analysis
26           $(Confidence, Intent) \leftarrow \text{Reanalyze}(C_1, C_2, InScope, Intent, Confidence, feedback)$ 
27           $iter \leftarrow iter + 1$ 
28       $justification \leftarrow \text{"No future commit shares patching intent with } C_1\text{"}$ 
29      return  $(\text{Good}, \emptyset, (justification, \emptyset))$ 

```

slices of C_1 and C_2 to obtain their shared impact region as the intent scope, then uses a second dedicated LLM to infer, within this scope, if C_2 shares C_1 's patching intent.

LLM-extracted intent—explicit or implicit—may still be hallucinated or otherwise erroneous. Phase 3 therefore validates the prior prediction via two *independent* checks: a judge model auditing whether the reasoning is coherent and grounded, and static analyzers summarizing vulnerability states across three snapshots (pre- C_1 , post- C_1 , post- C_2). If the two agree, the prediction is accepted; otherwise, bounded refinement is triggered up to a fixed budget (using any inconsistencies—either among the static analyzers, or between these analyzers and the LLM judge—as feedback).

If Phase 3 confirms C_2 as a future commit, PatchAudit produces the **Output**: it classifies C_1 as *Incomplete* or *Incorrect*, with a justification of why C_2 shares C_1 's patching intent and an issue-cause analysis of what made the initial patch deficient. These two bad-patch types are mutually exclusive, single-label annotation categories by design: If C_1 leaves the original vulnerability reachable, we consider it incomplete; if C_1 removes the original vulnerability but introduces a new security defect, we consider it incorrect. If no future commit is found, C_1 is classified as *Good* with a justification.

In accordance with this high-level overview, Algorithm 1 gives the step-level workflow. For large-scale measurement, PatchAudit analyzes an audited patch C_1 and stops at the first verified future commit, which provides evidence that C_1 is a manifestly bad patch. When Phase 1 does not yield a high-confidence explicit intent match, the pipeline enters Phase 2, which first computes the intent scope—the forward-slice intersection of C_1 and C_2 . If this intersection is empty, indicating that C_2 modifies no code region reachable from C_1 's patch, the candidate is classified as not a future commit and discarded without further analysis, as the absence of any shared code context precludes shared patching intent. Otherwise, the implicit intent model infers whether C_2 shares C_1 's patching intent within the scoped region. The feedback loop is cross-phase: when Phase 3 verification finds insufficient or conflicting support, it returns structured feedback to Phase 2 for selective re-analysis before verification is attempted again. For zero-day discovery, the same pipeline is reused without the early return at line 16, allowing the system to inspect the full post- C_1 history before checking whether the vulnerability persists in the latest version.

3.2 Explicit Intent Identification (Phase 1)

Phase 1 determines if C_2 *explicitly* signals an intent to continue the fix introduced by C_1 . Developers may communicate such relationships via commit-message references—CVE identifiers, patch hashes, issue links, or follow-up phrasing—but these signals vary. Phase 1 extracts and validates these signals as a lightweight pre-filter, aiming to confirm future commits without the deeper code analysis of Phase 2 (which is more expensive).

3.2.1 Potential Intent Extraction (Step 1.1). Phase 1 begins by identifying commits whose commit messages may indicate a connection to the initial fix C_1 . Developers refer to earlier patches or CVEs in highly variable and informal ways—ranging from explicit identifiers to vague phrases such as “follow up,” “continue previous fix,” or “handle remaining cases.” These references are also frequently used in unrelated contexts, including documentation edits, test updates, or reverts. Because this space of expressions is large, non-standardized, and noisy, PatchAudit uses a recall-oriented filter: any commit containing such references is treated as a *candidate* without interpreting the meaning of the reference. This choice is justified by that explicit follow-up fixes are relatively rare while missing one would compromise overall recall. By contrast, false positives here would merely incur a textual validation in Step 1.2 (Intent Validation). Explicit reference evidence includes CVE IDs, prior patch hashes, and issue/PR/advisory links. If no such evidence is found (i.e., extraction fails), the rest of Phase 1 is skipped and PatchAudit proceeds to Phase 2 for semantic analysis.

3.2.2 Intent Validation (Step 1.2). A reference alone does not establish that a commit continues the fix of C_1 : it may reflect a non-security change, a merge, or an incidental mention. Thus, Step 1.2 determines if the textual artifacts provide substantive evidence that C_2 addresses the same vulnerability and enforces the same security property as C_1 . These two aspects—context and property—capture the minimal structure of an explicit continuation. This decomposition provides a more reliable and interpretable account of continuation than a direct binary decision, which tends to conflate unrelated forms of textual similarity.

PatchAudit employs a compact text-only model for this task. Large end-to-end LLMs can easily over-interpret references, infer relationships that are not supported by the text, or be influenced by irrelevant contextual cues. By using a small model fine-tuned specifically on labeled reference cases, PatchAudit restricts the decision to the information actually present in commit messages and linked discussion. This avoids both the brittleness of pattern matching and the tendency of general LLMs to hallucinate intent.

The output of Step 1.2 is a continuation prediction accompanied by a confidence score, indicating how the textual evidence supports the claim that C_2 continues the fix of C_1 . When the confidence exceeds a threshold, the commit is marked as *text-confident* and routed to Phase 3. Otherwise, the case is treated as *text-uncertain* and escalated to Phase 2, where semantic evidence from code is used to re-evaluate the continuation question. This routing strategy leverages textual cues when they are reliable, while deferring ambiguous cases to the expressive code-based reasoning path.

All candidate (initial patch) commits first enter Step 1.1 (patching) intent extraction therein fails or the subsequent intent validation ends with low confidence, the commit is routed to Step 2.1.

3.3 Implicit Intent Analysis (Phase 2)

Phase 2 handles commits whose continuation intent is implicit in code rather than being explicit in text (commit log). It analyzes whether C_2 semantically extends C_1 by first identifying their shared behavioral region and then assessing whether the edits reinforce the same vulnerability context and security property.

3.3.1 Intent Scoping (Step 2.1). Many follow-up fixes contain no textual references. For instance, their intent is expressed only through modifications to callers, helper routines, or data/control-flow paths adjacent to—but not overlapping with—the lines patched in C_1 [26]. Such commits cannot be identified by Phase 1, and simple diff-based proximity would also be unreliable: the patching continuation may occur far from C_1 ’s edit location, while unrelated refactorings may appear textually adjacent.

To provide a stable semantic basis, PatchAudit computes inter-procedural forward slices as impact regions SC_1 and SC_2 , of C_1 and C_2 , respectively. These regions scope how each commit may influence downstream code behavior, including relevant callers, callees, data/control-flow paths, and security-sensitive sources and sinks. Their intersection, $InterSlice = SC_1 \cap SC_2$, captures where the two commits impact the same (vulnerable) code *if they address the same vulnerability*, even when they are in different functions or call-graph layers. *InterSlice* removes semantically disjoint edits and suppresses refactoring noise, since refactorings that are only textually adjacent but do not fall into the shared downstream impact region of both commits are excluded, whereas refactorings that

do affect the same vulnerability-relevant behavior remain in the intersection. It thus serves as the reduced evidence scope (hence mitigating LLM hallucinations) for Step 2.2. If no such shared semantic region exists, the candidate is discarded because there is no code-grounded basis for continuing with implicit intent analysis.

3.3.2 Intent Identification (Step 2.2). Within *InterSlice*, evidence of continuation is semantic and often subtle. A follow-up fix may tighten a boundary check, propagate a guard to a caller, or correct a misuse introduced by the initial patch. These patterns rarely align with textual similarity, so a one-shot classifier that ingests the entire *InterSlice* tends to rely on superficial overlap, overlook rare but decisive paths, and produce opaque predictions.

Thus, PatchAudit features an agentic module that reasons over *InterSlice* in a structured, goal-directed manner. Rather than producing a verdict in a single pass, it follows a *structured agentic workflow*: it decomposes the decision into concrete questions about the scoped region and identifies which fragments are relevant to answering them. Specifically, PatchAudit first presents *InterSlice* to the LLM as the initial evidence scope. The LLM reads this sliced code and, only when the current evidence is insufficient, asks to view more code. For instance, when understanding a modified function requires key callers, callees, or dependent checks, the LLM may ask to view that additional code rather than broadening scope indiscriminately. Unlike one-shot reasoning over a fixed context, it lets the model decide whether more code is needed while keeping expansion bounded. The output is a binary label, *future-commit* or *not-future-commit*, submitted to Phase 3 for validation.

3.4 Multi-Pronged Verification (Phase 3)

Phase 3 provides an independent validation layer for the predictions made in Phase 1 or Phase 2. Since an LLM-based agent may occasionally misinterpret a snippet, overlook context, or draw conclusions not fully supported by the extracted code, we do not directly trust its prediction. Instead, we seek confirmation from an external judge model and incorporate static vulnerability evidence derived from the underlying code base when available. These two checks (Steps 3.1 and 3.2) run in parallel and are then combined by an arbitration step (Step 3.3) to produce the Phase 3 decision.

3.4.1 Static Analyzer Verification (Step 3.1). Static analyzers can provide vulnerability evidence on snapshots before/after C_1 and C_2 . By comparing whether the detected vulnerability signals are preserved, removed, or newly introduced across these snapshots, PatchAudit derives an evidence-based notion of vulnerability continuity that can support or challenge the given prediction. Thus, Phase 3 orchestrates an ensemble of static analyzers, each independently checking the relevant snapshots and producing a vulnerability-state report. The resulting signals are then compared across key snapshots (e.g., before and after C_1 , and before C_2) to check if the same vulnerability evidence persists, disappears, or changes.

Since static analysis is imprecise and can be subject to false negatives as well (e.g., reporting a vulnerable snapshot as “not vulnerable”), PatchAudit treats this signal as corroborating evidence rather than a reliable oracle.

3.4.2 LLM-based Verification (Step 3.2). In parallel to the static analyzer-based verification, PatchAudit uses another LLM as an

independent judge to provide one further verification signal. This neural-symbolic, multi-pronged mechanism notably enhances the reliability of PatchAudit's final outputs as we later evaluated.

The judge model plays a strictly bounded role. It re-reads the code snippets referenced by the agent and checks whether the agent's chain of reasoning is logically self-consistent and grounded in the actual code inside *InterSlice*. This includes verifying that each cited snippet truly exists and that the agent's claims about same security context and same security property are supported by the evidence it selected. The judge does not generate new reasoning or propose alternative explanations; it produces only an *agree/disagree* decision together with a short audit trace summarizing its assessment.

3.4.3 Verdict Arbitration (Step 3.3). Phase 3 then combines the two parallel lines of evidence to validate the agent's current prediction. When the two lines provide mutual support, the prediction is accepted with high confidence. When they disagree or evidence is insufficient, PatchAudit does not immediately reject the prediction; instead, it triggers a bounded refinement loop that performs selective re-analysis guided by lightweight structured feedback. Concretely, this feedback summarizes the judge outcome (agree/disagree) together with which static consistency checks failed in the current iteration, prompting the agent to refine its reasoning and supporting evidence and, when warranted by the evidence, flip its prediction. The process is bounded by an iteration budget to prevent uncontrolled oscillation and to bound the time cost. If the budget is exhausted without obtaining sufficient validation, the module returns the agent's current prediction as a fallback final decision (without triggering an additional re-analysis step).

The result of arbitration is a structured decision with a binary label (future-commit or not-future-commit). To enable our in-depth measurement, the verification module also records the judge's audit trace and the supporting static vulnerability states, and attaches these records to the result. Based on these, Phase 3 generates the patch *classification* label and *explanation*, justifying the label and further classifying the issue and cause if the label is *Incomplete* or *Incorrect*, as PatchAudit's final outputs.

3.4.4 Holistic Workflow. Algorithm 2 formalizes the three steps above as a single verification step inside Phase 3. The top-level feedback loop lives in Algorithm 1: when verification cannot confidently accept the current prediction, it emits structured inconsistency feedback and hands control back to Phase 2 for selective re-analysis.

Algorithm 2 takes the current Phase 1/2 state (*Intent*, *Confidence*), materializes from it the agent's current prediction and supporting reasoning, and validates them using two independent signals: a judge audit over the agent's reasoning trace (line 3) and snapshot-level static vulnerability evidence (line 4). The verifier returns one of three outcomes: future-commit, not-future-commit, or unresolved. The third case indicates that the available evidence is insufficient or internally inconsistent, and therefore Phase 2 should refine the current semantic analysis before verification is attempted again.

The process begins by auditing the agent's reasoning. The judge takes (*agent_reasoning*, *agent_prediction*) and returns *judge_agree* $\in \{\text{agree}, \text{disagree}\}$ with respect to the *current* prediction (line 3). An agree verdict indicates that the reasoning trace is internally coherent and grounded in the provided snippets, while disagree indicates missing grounding or unsupported claims. The judge does

Algorithm 2: Holistic multi-pronged verification workflow

```

Input:  $C_1, C_2$ 
Input: Intent, Confidence: current prediction state from Phases 1/2
Output: verdict  $\in \{\text{future-commit}, \text{not-future-commit}, \text{unresolved}\}$ 
Output: feedback: lightweight signals for Phase 2 re-analysis
1 Function DualConsistencyValidate( $C_1, C_2, \text{Intent}, \text{Confidence}$ )
2   (agent_prediction, agent_reasoning)  $\leftarrow$  BuildPrediction(Intent, Confidence)
3   judge_agree  $\leftarrow$  AuditLogic(agent_reasoning, agent_prediction);
4   /* judge returns (agree, disagree) w.r.t. agent_prediction */
5   ( $C1\_pre, C1\_post, C2\_pre$ )  $\leftarrow$  GetVulnStates( $C_1, C_2$ )
6   feedback  $\leftarrow \emptyset$ ;
7   if agent_prediction = not-future-commit then
8     neg_v1  $\leftarrow$  (judge_agree = agree);
9     neg_v2  $\leftarrow$  CheckSafe( $C2\_pre$ );
10    neg_v3  $\leftarrow$  CheckDiffVuln( $C2\_pre, C1\_post$ );
11    if neg_v1 and (neg_v2 or neg_v3) then
12      return (not-future-commit,  $\emptyset$ )
13    feedback  $\leftarrow$  (judge_agree, neg_v2, neg_v3);
14    return (unresolved, feedback)
15  else if agent_prediction = future-commit then
16    pos_v1  $\leftarrow$  (judge_agree = agree);
17    pos_v2  $\leftarrow$  CheckSameVuln( $C1\_post, C2\_pre$ );
18    pos_v3  $\leftarrow$  CheckSameVuln( $C1\_pre, C2\_pre$ );
19    if pos_v1 and (pos_v2 or pos_v3) then
20      return (future-commit,  $\emptyset$ )
21    feedback  $\leftarrow$  (judge_agree, pos_v2, pos_v3);
22    return (unresolved, feedback)

```

not propose alternative reasoning or output a separate label; it only audits whether the agent's own evidence supports its prediction.

The verifier also retrieves snapshot-level static vulnerability states ($C1_pre, C1_post, C2_pre$) (line 4). These states summarize whether vulnerability evidence is detected at key snapshots around C_1 and C_2 . Because static evidence can be incomplete and subject to false negatives, the verifier treats it as corroborating evidence rather than a complete oracle.

The dual-consistency condition arises from combining the judge audit and the static evidence checks. The verifier branches on *agent_prediction* because the expected static pattern differs between the two labels. For not-future-commit (line 6), the verifier requires *judge_agree* = agree and then checks whether pre- C_2 is either safe (CheckSafe) or exhibits a different vulnerability state than post- C_1 (CheckDiffVuln). If this condition holds, the verifier accepts the prediction and returns not-future-commit (line 10). For future-commit (line 14), the verifier again requires *judge_agree* = agree and then checks whether pre- C_2 shares the same vulnerability state as post- C_1 or pre- C_1 (CheckSameVuln). If this holds, the verifier returns future-commit (line 18). Requiring judge agreement prevents accepting predictions whose reasoning is not grounded, while allowing multiple static evidence paths makes the rule robust to static-analysis false negatives.

When neither branch satisfies its dual-consistency condition, the verifier does not directly force a negative or positive decision. Instead, it returns unresolved together with lightweight structured feedback (line 13). Concretely, this feedback consists of the judge outcome and the failed static predicates in the current branch (e.g., (*judge_agree*, *pos_v2*, *pos_v3*) or (*judge_agree*, *neg_v2*, *neg_v3*)).

The caller, Algorithm 1, feeds this signal back into Phase 2 to refine the semantic analysis and then re-enters verification. This design makes the inconsistency loop explicit at the top level while keeping each verification step local and auditable. For transparency, the system can additionally record the judge audit trace, the supporting snapshot vulnerability states, and the per-iteration arbitration outcomes from the refinement loop.

Table 1: Statistics of BadPatch-Bench, including #positive (pos.) and negative (neg.) samples, and #unique CVEs with at least one positive sample (pos. CVEs)

Evidence scenario	#projects	#CVEs	#pos. CVEs	#pos. samples	#neg. samples
With explicit references	221	905	147	283	2,002
Without explicit references	143	242	80	80	1,753
All (4,118 samples)	265	1,147	227	363	3,755

4 BadPatch-Bench

Both the implementation (for fine-tuning the LLMs) and evaluation of PatchAudit require high-quality bad-patch datasets, which are currently unavailable. We therefore constructed *BadPatch-Bench*, covering both evidence scenarios of bad patches (§2.2).

With explicit references. We collected all CVEs from the NVD disclosed between 2012 and 2024 whose advisories link to a GitHub patch commit, and filtered for patch commits involving C/C++, Java, or Python code changes. For each qualifying patch C_1 , we retrieved all subsequent branch commits, excluding non-code-change commits (documentation, CI/CD changes, version bumps). We then identified CVEs where at least one remaining later commit references the original CVE or patch (via CVE id, commit hash, or issue link). For each such CVE, we labeled these later commits per their patching intent (§2.1), generating one (C_1, C_i) sample per commit.

Without explicit references. After selecting CVEs in the target languages, we filtered to repositories with at least 1,000 commits, a threshold that allows enough post- C_1 history for later intent-continuing commits—PatchAudit’s evidence for a bad patch—to appear; in smaller projects C_1 often sits near the end of observable history, leaving little opportunity for such evidence. This is a filter on the longitudinal-history observability of remediation history, not a security assumption about the repositories it excludes. We thus excluded CVEs in the *with-explicit-reference* scenario and randomly sampled from the remainder. For each C_1 , annotators examined subsequent branch commits chronologically, applying the patching-intent criterion to identify the first commit (if any) addressing the same vulnerability as C_1 .

Annotation protocol. All labels in both scenarios were first assigned by three annotators independently, with disagreements resolved via meetings/discussions to reach consensus. The inter-rater agreement Fleiss’ $\kappa = 0.77$, indicating strong labeling reliability.

Table 1 summarizes the resulting dataset. Each sample is a (C_1, C_i) commit pair where C_i is a candidate future commit; a single CVE may contribute multiple labeled pairs when its remediation chain $C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow \dots$ contains multiple bad-patch transitions. This is why the 363 positive samples correspond to 227 unique positive CVEs. The without-reference scenario yields a one-to-one CVE-to-sample mapping: annotators identified only the first future commit per C_1 due to the high cost of labeling. The with-reference scenario, by contrast, examines all later commits referencing C_1 , which can yield multiple positive pairs per CVE when the remediation chain involves several intent-sharing commits.

5 Implementation and Evaluation

5.1 Model Customization Via Fine-Tuning

PatchAudit features two task-specialized LLMs, each fine-tuned to reason about patch intent under different evidence conditions in Phase 1 and Phase 2. This customization serves two simultaneous

needs—task-specific accuracy on subtle, context-bound decisions, and analysis cost at scale—both justifiable because PatchAudit’s agentic design issues multiple sequential LLM calls per candidate commit across all phases, especially for large-scale measurement.

On accuracy, an off-the-shelf open-source model without our task-specific fine-tuning substantially degrades performance: removing Phase 2 fine-tuning drops F1 from 0.934 to 0.787, a 15-point reduction that reflects the difficulty of reasoning over semantic patching intent without task-specific supervision. On cost, replacing the fine-tuned models with a frontier LLM (GPT-5) for all three phases would raise the per-CVE cost (by 12.9×) to \$1.74, making our population-scale measurement study unaffordable to us and less practical to reuse. Fine-tuning therefore resolves both constraints at once, preserving the accuracy of dedicated task supervision at a fraction of the cost of frontier LLM inference.

5.1.1 Textual-Intent Model for Phase 1. This model is specialized to assess if a reference-bearing commit genuinely continues the fix intent of C_1 . Off-the-shelf LLMs fall short on this task due to ambiguous commit phrasing, project-specific conventions, and semantic overload of references (e.g., unrelated bookkeeping vs. fixes). Fine-tuning enables accurate disambiguation of true fix continuations. We fine-tune a compact transformer-based language model on labeled commit messages and their associated metadata (issues, pull requests, advisories), since commit messages provide the most direct statement of developer intent while the associated artifacts supply surrounding vulnerability context when the message itself is brief or ambiguous. We deliberately exclude code diffs from this input, as the code-level relationship between C_1 and C_2 is the subject of Phase 2; including code here would conflate the two phases and undermine the efficiency motivation for the two-phase split. The model outputs a binary label: `continues` vs. `does_not_continue`. We apply a confidence threshold over the model’s output probabilities to flag high-certainty cases for direct resolution in Phase 1, and defer uncertain (i.e., low-confidence) cases to Phase 2 for deeper semantic analysis.

5.1.2 Semantic-Intent Model for Phase 2. This model is specialized to determine if C_2 corrects or completes the fix attempted by C_1 , which requires interpreting semantic evidence inside *InterSlice* between C_1 and C_2 . Off-the-shelf LLMs often hallucinate or overlook subtle security semantics across commits. Fine-tuning trains the model to identify residual vulnerabilities and assess fix continuity.

We fine-tune a code-capable LLM to operate over the *InterSlice*, along with the CVE/CWE description and the respective code diffs. Training examples are accompanied by process-style reasoning traces, which capture the chain of justification: what security intent was present in C_1 , what behavior remains unpatched, and how (or whether) C_2 completes or corrects it (or why not). This supervision enables the model to produce explanations grounded in specific code regions, aligned with PatchAudit’s verification logic. At inference time, the model emits its reasoning trace and prediction outcome in a single pass, with each reasoning step required to cite code fragments retrieved within the *InterSlice*.

5.2 Tool Implementation

Phases 1 and 2 fine-tune Qwen3-14B with LoRA on 85% of BadPatch-Bench’s samples with and without explicit references, respectively.

Table 2: PatchAudit effectiveness vs. baselines; best baseline performance is in bold; subscripts denote standard error

Model	Precision	Recall	F1	Micro-F1	Macro-F1
PatchAudit	.970 $\pm$.007	.941 $\pm$.007	.955 $\pm$.006	.885 $\pm$.003	.813 $\pm$.007
GPT-5.2	.962	.556	.704	.746	.624
Claude-Opus-4.5	.889	.711	.790	.755	.638
Gemini-3-Pro	.839	.578	.684	.727	.628
DeepSeek-R1	.781	.556	.649	.682	.564
Qwen3-Coder	.818	.600	.692	.718	.609
GPT-oss:20B	.700	.467	.560	.682	.568
CodeQL-based static detector	1.000	.044	.085	.609	.312

Table 3: Stage-wise validation, per commit pair. Phase 1 only routes and makes no terminal rejections.

Stage/phase	#Rejected pairs	False rejects%	Rejection accuracy
Phase 1	0	—	—
Phase 2	510	5/510=1.0%	505/510=99.0%
Phase 3	12	2/12=16.7%	10/12=83.3%

Phase 1 uses 0.85 as confidence threshold. Phase 2 uses Joern [9] to compute forward slices. Phase 3 uses GPT-5 as the judge model and two complementary static analyzers: CodeQL and Semgrep.

5.3 Evaluation

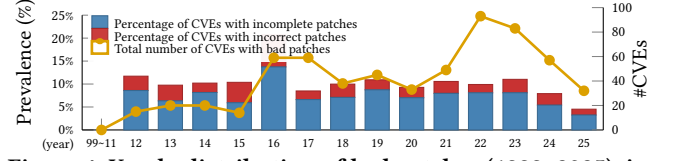
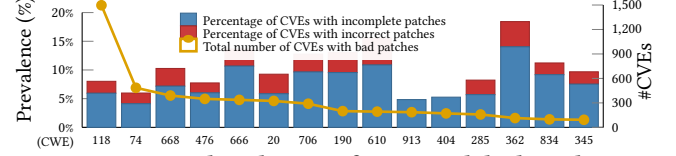
5.3.1 Setup. We use the remaining 15% of BadPatch-Bench, ensuring no project- or CVE-wise overlap with the fine-tuning data, for evaluation, including 110 initial patches (45 bad, 65 good) across 62 projects, constructed from 565 candidate (C_1, C_i) commit pairs.

All experiments ran on Ubuntu 24 with an NVIDIA A6000 for Phases 1 and 2 inference, with mean $\pm$ stderr reported over 3 runs.

5.3.2 Effectiveness. Table 2 shows PatchAudit’s effectiveness on bad-patch detection. It achieves 95.5% F1 in detecting bad patches (97.0% precision, 94.1% recall) over the 110 initial patches in the evaluation set, and additionally reaches 88.5% micro-F1 for joint detection and three-way classification into Good, Incomplete, or Incorrect. As shown, standard errors are below 0.01 across all metrics. Yet, these numbers only reflect run-to-run stability.

To further assess the uncertainty of our evaluation, we performed a nonparametric bootstrap over the stored outputs for the 110 patches (10,000 resamples), resampling by initial patch/CVE rather than by candidate pair—to avoid treating multiple pairs from the same CVE as independent. We obtained 95% confidence intervals of F1 .955 [.903, .990], precision .977 [.921, 1.00], and recall .933 [.854, 1.00], capturing case-level sampling uncertainty, with the F1 lower bound still above the best baseline (.790). Ablations show that PatchAudit’s performance is contributed notably by each of its components, mostly by Phase 2, followed by intent scoping and Phase 2 fine-tuning.

5.3.3 Stage-wise validation. Beyond end-to-end effectiveness, we validate each phase separately against ground truth on the evaluation set (Table 3). Phase 1 is a routing stage: candidates without confident textual evidence escalate to Phase 2 rather than being rejected, so it introduces no terminal false negatives. Phase 2 provides the main semantic decision, rejecting 510 candidates with only 5 false rejects (1.0%; 99.0% reject accuracy). Phase 3 then verifies and refines these decisions, rejecting 12 predictions of which 2 are erroneous (16.7%; 83.3% reject accuracy); it is net-corrective, removing false positives and recovering true positives that Phase 2 had rejected, so errors do not compound across phases. These counts are per commit pair, aggregated per patch.

**Figure 4: Yearly distribution of bad patches (1999–2025), in terms of prevalence (%), left axis and total count (right axis).****Figure 5: CWE distribution of CVEs with bad patches.**

5.3.4 Baseline Comparison. We compare PatchAudit against GPT-5.2, Claude Opus 4.5, Gemini-3-Pro, and a CodeQL-based static detector. We do not include patch-verification methods as baselines because they target a different task from ours and are not readily adaptable to bad-patch detection. For example, VeriBin [32] checks functional equivalence between patched and unpatched binaries, rather than whether a patch leaves the target vulnerability unresolved or introduces a new security defect. We also do not include CoLeFunDa [38] or LLM4VFD [35] as baselines, because either no usable public implementation is available or substantial adaptation would be required to repurpose them for our task.

The CodeQL baseline runs the same security queries on both the pre- C_1 and post- C_1 snapshots and compares their vulnerability evidence: if the post- C_1 snapshot still exhibits the same vulnerability signal as pre- C_1 , or if new vulnerability signals appear only after applying C_1 , the baseline classifies C_1 as a bad patch.

PatchAudit notably outperforms all baselines (Table 2), including Claude Opus-4.5 (the best baseline) by 16.5 points higher in F1. CodeQL attains perfect precision but very low recall (F1=0.085), capturing only syntactic or flow-level matches and missing the semantic follow-ups that dominate bad patches in our study.

5.3.5 Cost. PatchAudit takes 534.86 seconds and \$0.025 per commit, or 2,799 seconds and \$0.135 per CVE end-to-end. Its monetary cost is comparable to the strongest LLM baseline (Claude Opus-4.5: \$0.126 per CVE), while offering an F1 improvement by 20%.

Together, the overall results, component ablations, stable reruns, stage-wise validation, and low per-CVE cost show that PatchAudit is reasonably accurate, robust, and affordable enough to serve as the measurement instrument for our large-scale study.

6 Measurement Study

We applied PatchAudit as a measurement instrument to conduct a large-scale empirical study of bad security patches.

Study population. Our study population $\mathcal{P}$ consists of all 6,535 CVEs disclosed in NVD between 1999 and 2025 whose advisories link to a GitHub-hosted patch commit in C/C++, Java, or Python, spanning 1,649 repositories. We analyze $\mathcal{P}$ as a census via two non-overlapping paths that partition it ($389 + 6,146 = 6,535$).

Fully-annotated subset (389 CVEs). BadPatch-Bench (§4; covering 1,147 CVEs in total) provides end-to-end annotation for 389 of them: 147 with-reference positives (annotators only searched for commits that textually referenced the initial patch) and 242

without-reference CVEs (annotators inspected every later commit exhaustively). These 389 yield 227 confirmed bad-patched CVEs.

Audit corpus (6,146 CVEs). The remaining 758 BadPatch-Bench CVEs are with-reference *negatives*: annotators ruled out reference-bearing future commits but never inspected for implicit ones, so they re-enter the audit pool together with the 5,388 CVEs of $\mathcal{P}$ outside BadPatch-Bench ($5,388 + 758 = 6,146$). PatchAudit scans this corpus and flags 433 candidates. Three annotators independently confirmed each flagged case against the code, diffs, and available CVE/commit evidence rather than accepting the tool’s predicted label; although the flag was visible, labels were assigned from this evidence. This confirms 390 cases as true positives (90.1% precision) and overrules PatchAudit on the other 43 cases ($\sim 10\%$), with Fleiss’ $\kappa = 0.71$ on the three-way classification—indicating the confirmation was not a rubber-stamp process. The census yields $227 + 390 = 617$ bad patches in total; all reported rates in this section are population-level over $\mathcal{P}$.

6.1 Prevalence and Distribution

Of the 617 bad patches, 457 (74.1%) are incomplete and 160 (25.9%) incorrect, for an overall bad-patch prevalence of $617/6,535 = 9.4\%$ in $\mathcal{P}$ (7.0% incomplete, 2.45% incorrect). The incomplete rate aligns with Li et al.’s $\sim 7\%$ estimate on a much smaller manually labeled sample [13], while the overall 9.4% exceeds Woo et al.’s 5.9% for unreliable patches [31]. The gap likely reflects a broader evidence model: only 221 of the 617 cases (35.8%) have a follow-up commit that explicitly references the original CVE or patch; the rest (64.2%) are surfaced only via semantic code analysis (Table 4)—a category invisible to prior keyword- or hash-based linkage approaches.

Figure 4 shows the yearly prevalence. Non-zero counts in 1999–2011 confirm bad patches have been present since early CVE tracking, though sample sizes there are too small for reliable rates. From 2012–2024 (annual CVE count > 100), the bad-patch rate ranges from 7.9% (2024) to 20.8% (2016); the absolute count peaks at 93 in 2022 and declines through 2025 due to shorter observation windows for recent CVEs (most pronouncedly 2025’s artificially low 4.5%). The 2020–2024 pooled rate (9.7%) matches the overall 9.6%.

Bad patches affect nearly 1 in 10 CVEs and they persist over time. Incomplete patches are nearly 3 $\times$ prevalent as incorrect ones, and both types are mostly silently amended without referencing original CVE or patch.

We normalize CWE labels to the first level below the root in the CWE-1000 Research View [19], rolling descendant CWEs upward; 75 CVEs labeled at the root itself were reassigned downward by three annotators (Fleiss’ $\kappa = 0.79$). Figure 5 shows that CWEs requiring non-trivial invariants over values, state, or naming exhibit the highest bad-patch rates: Concurrent Execution (CWE-362), Externally Controlled Reference (CWE-610), Resource Lifetime (CWE-666), Integer Overflow (CWE-190), and Incorrectly-Resolved Name (CWE-706) all exceed 12%. By contrast, Indexable Resource Access (CWE-118) and Special Elements Neutralization (CWE-74) dominate in absolute volume—together nearly one third of the population—yet show only moderate bad-patch rates. Incomplete patches outnumber incorrect ones across every CWE category.

The most patch-failure-prone vulnerabilities are not the most common. CWEs involving concurrency and resource lifetimes show the highest bad-patch rates, while high-volume memory-access CWEs are patched more reliably.

Table 4: Characteristics of bad patches of Since-2020 (2020–2025) vs. Before-2020 ($\uparrow$ increase, $\downarrow$ decrease); time in days

Metric	Before 2020		Since 2020		All Years	
	Explicit	Implicit	Explicit	Implicit	Explicit	Implicit
Bad-patch count	140	130	81 $\downarrow 42\%$	266 $\uparrow 105\%$	221	396
Bad-patch prevalence (%)	5.7	5.3	2.1 $\downarrow 63\%$	6.7 $\uparrow 26\%$	3.5	6.2
Total #patches	2.5	2.3	2.5 $\downarrow$	2.5 $\uparrow 9\%$	2.5	2.4
#Changed files per patch	4.0	1.9	2.5 $\downarrow 38\%$	3.0 $\uparrow 58\%$	3.4	2.7
#Changed functions per patch	7.6	5.4	6.4 $\downarrow 16\%$	7.7 $\uparrow 43\%$	7.2	7.0
#Changed lines per patch	50.5	44.4	103.9 $\uparrow 106\%$	84.0 $\uparrow 89\%$	70.1	72.5
Time from CVE (disclosure) to initial patch	-192.9	-165.5	-163.4 $\uparrow 15\%$	-106.1 $\uparrow 36\%$	-182.1	-125.6
Time from initial patch to future commit	424.8	471.5	136.9 $\downarrow 68\%$	203.7 $\downarrow 57\%$	319.3	281.6
Time from CVE (disclosure) to future commit	232.3	380.1	-26.0 $\downarrow 111\%$	98.0 $\downarrow 74\%$	137.7	180.0

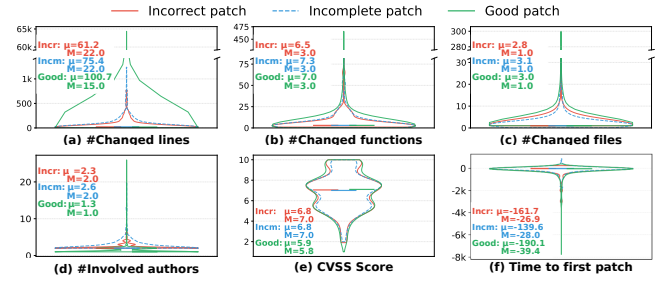


Figure 6: Distribution of good vs. bad patches across change scope (a-c), vuln. severity (d), and response timing (f), including mean (μ , shown by horizontal bars) and median (M).

6.2 Scope, Severity, and Response Timing

Temporal evolution of bad-patch characteristics. Modern vulnerability management has shifted substantially since coordinated disclosure became standard practice; we ask whether bad-patch characteristics have shifted accordingly. We use 2020 as a coarse, illustrative boundary for contrasting earlier and more recent practice, summarizing a gradual trend rather than marking an abrupt change at a specific year. Splitting the 617-case set at this boundary (Table 4) shows a notable shift: explicit bad patches—those whose follow-up commits textually reference the original CVE or patch—declined from 51.9% to 23.3% of cases, while implicit ones rose to 76.7%. Remediation also accelerated 2–3 $\times$, and for explicit cases the first follow-up commit lands on average 26 days *before* CVE disclosure, vs. before 2020 it arrived 232 days *after*.

Bad patches have become faster—but quieter. As remediation accelerates, transparency declines—making silent bad patches the new norm.

Good vs. bad patches. Are bad patches distinguishable from good ones on observable attributes? Figure 6 compares the two across code-change scope (a–c), involved authors (d), vulnerability severity (e), and response timing (f). CVSS severity is virtually identical across groups and medians for code-change scope also overlap, meaning the *typical* bad patch is indistinguishable from a good one. The differences lie in the tails: good patches exhibit a much heavier tail in code-change scope and in response timing, while bad patches cluster in smaller changes with shorter pre-disclosure windows. The timing pattern is consistent with coordinated disclosure granting more development time, though alternative explanations cannot be excluded without case-level investigation. Bad patches are typically amended by a different developer (median 2 authors vs. 1 for good patches), not self-corrected by the original author.

```

Commit message: fixed #2364
Commit ID: 3a2458a49b3e6399709d456d7b35e7a6f50cfb86
File Path: src/bifs/unquantize.c
1 + if (NbBits > 32)
2 + return GF_NON_COMPLIANT_BITSTREAM;
3 // later: val = 1 << (NbBits - 1); NbBits==0 produces 1<<-1

Commit message: fix integer overflow issue in bifs/unquantize.c:298
Commit ID: 16c4f9c2881112eba7051cac48f922eb2b94e06
File Path: src/bifs/unquantize.c
1 - if (NbBits > 32)
2 + if (!NbBits || NbBits > 32)
3     return GF_NON_COMPLIANT_BITSTREAM;

```

Figure 7: CVE-2023-23144 (gpac): integer overflow via $1 \ll (\text{NbBits} - 1)$. Missing Boundary Values: C_1 (top) guards $\text{NbBits} > 32$ but omits $\text{NbBits} == 0$; C_2 adds the zero check.

```

Commit message: no 'OP_HASHADD' required when 'val' is false.
Commit ID: b611c43a5de061ec21b343967e1b64c45c373d7e
File Path: mrbgems/mruby-compiler/core/codegen.c
1 if (update) {
2 - if (len > 0) {
3 + if (val && len > 0) { // patched
4     pop_n(len+2+1);
5     genop_2(s, OP_HASHADD, cursp(), len);

Commit message: codegen.c: fixed hash codegen for 'lval'.
Commit ID: ae3c99767a27f5c6c584162e2adc6a5d0eb2c54e
File Path: mrbgems/mruby-compiler/core/codegen.c
1 - if (len > 0) {
2 + if (val && len > 0) { // now patched
3     pop_n(len+2);
4     genop_2(s, OP_HASH, cursp(), len); ...
5 - if (len > 0 || update) {
6 + if (val && (len > 0 || update)) {
7     pop(); pop();
8     genop_1(s, OP_HASHCAT, cursp());

```

Figure 8: CVE-2022-0326 (mruby): NULL dereference in hash code generation. Missing Parallel Locations: future commit applies the same guard to the two additional code paths.

```

Commit message: Fix #154/#155 padding check logic..
Commit ID: 76c9alfdf19d9e92e566a77376673e522aae9f80
File Path: src/cp/relic_cp_rsa.c
1 int r = 0;
2 for (int i = 0; i < len; i++) {
3     pad = (uint8_t)t->dp[0];
4     r |= pad ^ id[len - i - 1]; // r==0 iff match
5     bn_rsh(t, t, 8);
6 + if (r && m_len > 0 && counter >= 8) {
7     result = RLC_OK; // WRONG: accepts when r!=0

Commit message: Fix RSA verification rejecting correct IDs.
Commit ID: 2e02f61196911c7d0df388cce3a8419520a0dfc
File Path: src/cp/relic_cp_rsa.c
1 - if (r && m_len > 0 && counter >= 8) {
2 + if (r == 0 && m_len > 0 && counter >= 8) {
3     result = RLC_OK; } // now accepts when r==0

```

Figure 9: CVE-2020-36316 (RELIC): PKCS#1 v1.5 signature verification bypass. Logic Inconsistency: C_1 (top) accepts when $r \neq 0$ (mismatch); C_2 (the future commit) corrects to $r == 0$.

Incomplete vs. incorrect patches. Do the two failure modes differ from each other? Severity and author count are virtually identical between them, and medians overlap on every dimension—the typical incomplete patch is similar to the typical incorrect one. In means, incomplete patches are modestly but consistently larger across all scope metrics, while incorrect patches show slightly longer mean pre-disclosure lead time. Both gaps are small, though.

The typical bad patch is indistinguishable from a good one on every observable attribute; differences emerge only in distributional tails. Incomplete and incorrect patches are even harder to separate. No surface signal—size, severity, timing, or author count—is sufficient for detection or triage, reinforcing the necessity of deeper, code-level semantic analysis.

6.3 Failure-Cause Patterns

To understand why the bad patches fail, we classify each of them as per our two-level failure-cause taxonomy—5 top-level categories

(L1) and 19 sub-patterns (L2)—developed via grounded theory coding [2]. Categories emerged from iterative coding by three independent annotators, with Fleiss’ κ of 0.84 for L1 and 0.83 for L2, indicating almost perfect agreement. Each case is assigned to exactly one sub-pattern. The codebook and coding details, including decision rules for borderline cases, are in our artifact package.

Theme 1: Under-Scoped Remediation (63.4%). The fix is conceptually correct but does not reach all conditions under which the vulnerability is exploitable. The two L1 categories in this theme differ in scope: *Edge Case Missed* involves missing a specific value or state within a single attack vector, while *All Scenarios Not Considered* involves leaving some attack vectors entirely unaddressed.

Edge Case Missed (196 cases, 31.8%). The dominant sub-pattern, *Missing Boundary Values* (88), enforces correct behavior over the interior of the valid input domain but omits domain-boundary values such as zero, type-maximum, or empty inputs. Figure 7 gives an example: the initial patch added an upper-bound guard to prevent a left-shift overflow, but did not consider the specific value (0), which produces undefined behavior via $1 \ll -1$. The second sub-pattern, *Uncovered Input Variants* (56), captures patches that sanitize one syntactic form of a malicious input but miss canonically equivalent representations—alternate encodings, URL-percent-encoding, Unicode normalization, or path aliases—that bypass the check. The third, *Overlooked Exceptional States* (52), is where the patch secures the primary control-flow path but omits alternative paths activated by exceptional conditions—error returns, signal handlers, timeouts, or early exits—that converge on the same vulnerable operation.

All Scenarios Not Considered (195 cases, 31.6%). The dominant sub-pattern is *Missing Parallel Locations* (109)—the largest in the entire taxonomy, where identical vulnerable logic exists at multiple sites and the patch remediates only one. As Figure 8 shows, the initial patch (C_1) added a `val &&` guard to `OP_HASHADD` to prevent a NULL dereference, but `OP_HASH` and `OP_HASHCAT`—two independent branches performing the identical stack operation—were left unguarded. Another sub-pattern, *Missing Execution Paths* (61), is where the vulnerable operation resides at a single site but multiple paths reach it, and only one inbound path is guarded. The smaller sub-patterns—*Platform or Architecture Considerations* (10), *Resource Lifecycle Handling* (9), and *Protocol Semantic Variants* (6)—each reflect attack vectors the developer did not consider.

Most bad patches fail by stopping too early, not by fixing the wrong thing: they handle the obvious case correctly, but miss boundary values, equivalent input forms, exceptional paths, or parallel code sites. The dominant role of missing parallel locations suggests that the failure is often in (incomplete) attack-surface enumeration, not local patch correctness.

Theme 2: Flawed Security Realization (36.6%). The developer’s understanding of the vulnerability is adequate, but the code does not faithfully implement that understanding. The three L1 categories differ in where the mismatch occurs: the fix mechanism, the guard predicate, or the inter-component data and state propagation path.

Incorrect Implementation (117 cases, 19.0%) covers cases where the patch correctly identifies the vulnerability but introduces a defect in the remediation code itself. The dominant sub-pattern is *Logic Inconsistency* (71), where the patch introduces a Boolean logic error—an inverted predicate, incorrect operator, or reversed comparison—such that the defensive expression evaluates to the

```

Commit message: [2.2.x] Tightened file-upload path sanitation.
Commit ID: 04ac1624bdc2fa73718840175cf95ced122d26d
File Path: django/core/files/utils.py
1 + def validate_file_name(name):
2 +     if name != os.path.basename(name): // rejects / or \
3 +         raise SuspiciousFileOperation(...)

File Path: django/db/models/fields/files.py
1 + filename = validate_file_name(filename) // BEFORE upload_to
2 if callable(self.upload_to):
3     filename = self.upload_to(instance, filename)

Commit message: Relaxed file name validation in FileField.
Commit ID: b55699968fc9ee985384c64e37f6cc74a0a23683
File Path: django/core/files/utils.py
1 + def validate_file_name(name, allow_relative_path=False):
2 + if allow_relative_path:
3 +     path = pathlib.PurePosixPath(name)
4 + if path.is_absolute() or '..' in path.parts:
5 +     raise SuspiciousFileOperation(...) // only .. blocked

File Path: django/db/models/fields/files.py
1 if callable(self.upload_to):
2     filename = self.upload_to(instance, filename) ...
3 + filename = validate_file_name( // AFTER upload_to
4 + filename, allow_relative_path=True)

```

Figure 10: CVE-2021-31542 (Django): directory traversal via uploaded filenames. *Overly Strict or Broad*: C_1 (top) rejects any path separator before `upload_to()`; C_2 narrows to .. segments only and moves the check after `upload_to()`.

Table 5: Distribution of causes of bad patches, with L1 (in boldface) and L2 (sub-patterns) categories

Failure cause pattern / Sub-pattern	Incomplete	Incorrect	All bad patches
Edge Case Missed	163 (35.7%)	33 (20.6%)	196 (31.8%)
Missing Boundary Values	73 (16.0%)	15 (9.4%)	88 (14.3%)
Uncovered Input Variants	50 (10.9%)	6 (3.8%)	56 (9.1%)
Overlooked Exceptional States	40 (8.8%)	12 (7.5%)	52 (8.4%)
All Scenarios Not Considered	169 (37.0%)	26 (16.2%)	195 (31.6%)
Missing Parallel Locations	93 (20.4%)	16 (10.0%)	109 (17.7%)
Missing Execution Paths	54 (11.8%)	7 (4.4%)	61 (9.9%)
Platform/Arch. Considerations	8 (1.8%)	2 (1.2%)	10 (1.6%)
Resource Lifecycle Handling	8 (1.8%)	1 (0.6%)	9 (1.5%)
Protocol Semantic Variants	6 (1.3%)	0 (0.0%)	6 (1.0%)
Incorrect Implementation	50 (10.9%)	67 (41.9%)	117 (19.0%)
Logic Inconsistency	41 (9.0%)	30 (18.8%)	71 (11.5%)
Wrong Variable/Pointer Usage	3 (0.7%)	22 (13.8%)	25 (4.1%)
API Misuse	6 (1.3%)	15 (9.4%)	21 (3.4%)
Incorrect Conditioning	52 (11.4%)	31 (19.4%)	83 (13.5%)
Overly Strict or Broad	27 (5.9%)	16 (10.0%)	43 (7.0%)
Check Wrong Position/Order	12 (2.6%)	11 (6.9%)	23 (3.7%)
Wrong Value/Threshold/Constant	11 (2.4%)	3 (1.9%)	14 (2.3%)
Off-by-One	2 (0.4%)	1 (0.6%)	3 (0.5%)
Data Path Inconsistency	23 (5.0%)	3 (1.9%)	26 (4.2%)
Resource Lifecycle Mismatch	11 (2.4%)	0 (0.0%)	11 (1.8%)
Inconsistent State Across Contexts	7 (1.5%)	2 (1.2%)	9 (1.5%)
TOCTOU/Timing	4 (0.9%)	1 (0.6%)	5 (0.8%)
Incorrect Pointer Reference	1 (0.2%)	0 (0.0%)	1 (0.2%)
Total	457 (74.1%)	160 (25.9%)	617 (100.0%)

opposite of the intended result. In the Figure 9 example, the initial patch restructured the logic so that the per-byte XOR accumulator `r` should equal zero when the padding bytes match; however, the condition is mistakenly written (line 6) to accept the signature when `r` is non-zero—inverting the verification semantics. *Wrong Variable or Pointer Usage* (25) and *API Misuse* (21) together account for the remaining cases, capturing patches that reference the wrong identifier or call the wrong library function despite correct intent.

Incorrect Conditioning (83 cases, 13.5%) covers cases where a guard construct is present and reached during execution, but its predicate is incorrectly specified, causing the guard to admit unsafe inputs or reject safe ones. Unlike *Incorrect Implementation*, where the fix mechanism itself is wrong, here the failure is in *when* the guard permits or blocks execution rather than in *what* the fix does. The dominant sub-pattern is *Overly Strict or Broad* (43), where the guard fires at the correct program point but its acceptance region is incorrectly calibrated. In the Figure 10 example, the initial patch added a `validate_file_name()` routine that rejected

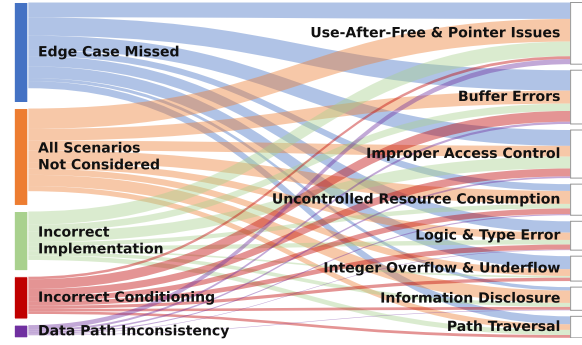


Figure 11: Mapping from L1 failure-cause categories to their most frequently associated (non-exhaustive) issue categories.

any filename for which `name != os.path.basename(name)`, catching path separators intended for traversal; however, the check was applied to the entire generated filename rather than only its terminal component, which caused legitimate `upload_to()` return values such as `uploads/2021/file.jpg` to be rejected alongside traversal attempts. *Check at Wrong Position or Order* (23) and *Wrong Value, Threshold, or Constant* (14) capture the remaining conditioning failures—guards that fire after the unsafe operation has already executed, or guards parameterized with incorrect literals.

Data Path Inconsistency (26 cases, 4.2%) covers cases where the patch is locally correct in isolation but fails because the developer did not account for how data, memory ownership, or shared security-relevant state propagates across component boundaries. The dominant sub-pattern is *Resource Lifecycle Mismatch* (11), where the patch introduces, or fails to correct, a resource-management error arising from incorrect ownership tracking across divergent execution branches. In CVE-2023-25136 (OpenSSH), the initial patch added a `free()` call without tracking pointer ownership across conditional branches, producing a double-free. *Inconsistent State Across Contexts* (9) and *TOCTOU/Timing* (5) capture the remaining cases, reflecting state-propagation failures and temporal gaps between validation and enforcement, respectively.

Flawed realization accounts for the remaining third of bad patches—they fail in translating intent to code: the developer knows what to check, but encodes the wrong check, places it incorrectly, or breaks invariants across component boundaries. Logic Inconsistency is the dominant sub-pattern, where defensive predicates evaluate opposite to the intended condition—a failure visible only through semantic reasoning about patch logic, not structural inspection.

Incomplete vs. Incorrect Patches. Table 5 overviews the failure causes and reveals key differences between the two failure modes. Incomplete patches are dominated by coverage failures: 72.7% fall under Theme 1, while incorrect patches instead concentrate in realization failures: 63.2% fall under Theme 2. The sharpest divergences are *Wrong Variable or Pointer Usage* and *API Misuse*—indicating that incorrect patches typically arise from developers reaching for the wrong primitive rather than misjudging scope. This contrast supports treating incomplete and incorrect patches as distinct remediation failure modes, not one undifferentiated class of bad fixes.

6.4 Issue–Cause Associations

To understand which failure causes lead to which types of issues—i.e., resurfaced (unresolved or newly introduced) vulnerabilities due

Table 6: Severity of defects introduced in incorrect patches

Severity	Description/Explanation	Count	%
High	Directly exploitable: authentication bypass, memory corruption, arbitrary code execution, or privilege escalation	69	43.1
Medium	Security-relevant but indirect: denial of service, crash, information disclosure, or logic errors with security impact	73	45.6
Low	Functional or resource issue with no direct security impact: memory leak, broken feature, or incorrect output	18	11.2

to the patch failures, we further classified each bad patch by its issue type as per our 13-category taxonomy in our artifact.

Figure 11 maps each L1 cause to its most frequently associated issue categories, revealing a clear asymmetry between *coverage failures* and *realization failures*. The two under-scoped causes, *Edge Case Missed* and *All Scenarios Not Considered*, spread broadly across issue categories. This suggests that under-scoping is largely vulnerability-agnostic: once a patch covers only part of the exploitable surface, the residual issue may reappear across many vulnerability classes, depending on the original vulnerability.

By contrast, flawed-realization causes exhibit sharper issue-level concentration. *Incorrect Implementation* is strongly associated with *Use-After-Free & Pointer Issues*, indicating that wrong variables, wrong pointers, and API misuse are especially damaging in memory-management code. *Incorrect Conditioning* concentrates in *Buffer Errors* and *Improper Access Control*, consistent with guards whose predicates are too broad, too narrow, inverted, or placed at the wrong program point. *Data Path Inconsistency*, although the smallest cause category, is disproportionately tied to pointer/lifetime and concurrency-related issues, reflecting failures in ownership, shared state, or time-sensitive propagation across component boundaries.

Under-scoped causes spread broadly across issue types, showing that incomplete coverage is a general bad-patch failure cause across vulnerability classes. Flawed-realization causes are more concentrated: implementation mistakes cluster around pointer/lifetime issues, while misconditioned guards cluster around buffer and access-control failures.

6.5 Severity of Incorrect Patches

Incorrect patches remove the original vulnerability but introduce new defects into the software. To characterize the security impact of these defects, we classify each of the 160 incorrect patches along a three-tier severity scale, as Table 6 shows. 88.8% of the introduced defects carry substantive security impact (High or Medium tier). Within the High-severity tier, *use-after-free* and *authentication bypass* are the two most frequent outcomes. The Medium-severity tier is dominated by *denial-of-service* and *crash* defects.

An example high-severity incorrect patch is the initial patch for CVE-2010-4250 in the Linux kernel `inotify` subsystem. The original bug was a memory leak in `inotify_init1`, triggered when `anon_inode_getfd` failed to allocate a file descriptor. The initial patch (a2ae4cc9) attempted to fix this leak by adding a single `fsnotify_put_group(group)` call on the error path. However, the same path already contained `free_uid(user)`; after the new group release, this call double-freed the `struct user` reference owned by the group. As a result, the patch transformed a memory leak into a locally triggerable kernel double-free, potentially exploitable by unprivileged users for local privilege escalation. The follow-up patch (d0de4dc5) corrected the ownership model by consolidating the user reference into the group lifecycle and removing the redundant manual `free_uid`.

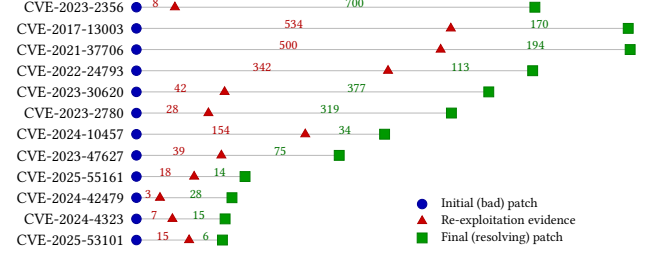


Figure 12: Exposure window of bad patches with re-exploitation evidence; numbers are #days on a square-root scale (drawn lengths are not proportional to elapsed time).

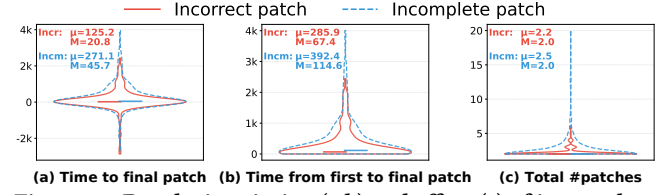


Figure 13: Resolution timing (a,b) and effort (c) of incomplete vs. incorrect patches, including mean (μ , shown by horizontal bars) and median (M).

Incorrect patches rarely trade security for harmless bugs. In the majority (88.8%) of cases, they introduce a new security-impacting defect, and 43.1% create exploitable flaws such as use-after-free or authentication bypass.

6.6 Bad-Patch Exploit Evidence

Our study examines bad patches retrospectively; thus, typically the bad patches have now been amended eventually. However, the affected software remains vulnerable between an initial (bad) patch and its final follow-up commit (*final resolving patch*)—a window in which the flawed fix is the only available remediation. To assess whether this window is exploited in practice, we searched Exploit-DB [25] and GitHub Security Advisory [6] for public exploits targeting the surface left open by the bad patch. Across the 617 bad patches, we identified 12 cases that meet this criterion.

As Figure 12 shows, bad patches create real exposure windows. Public re-exploitation evidence can appear quickly—sometimes within days of the initial patch—showing that the residual flaw may be readily reachable. In other cases, evidence emerges only after months, indicating that bad patches may remain latent for a while. Importantly, exploit evidence does not guarantee rapid remediation: the gap from public evidence to the final corrective patch can be substantial, and in one case extends nearly two years.

Bad patches are not merely retrospective failures; they can turn remediation into exposure, creating a misleading state of “patched but still exploitable”.

6.7 Resolution Timing and Effort

We next examine the resolution cost of bad patches: how long they remain unresolved and how many corrective commits are needed before convergence. As Figure 13 shows, incomplete patches exhibit heavier resolution cost. They take longer to reach the final resolving patch after CVE disclosure, and the gap is even clearer when measured from the first (bad) patch to the final amendment. The total #patches is similar at the median, but incomplete fixes require slightly more commits on average.

Table 7: Zero-day bad patches discovered by PatchAudit and confirmed by maintainers. Following responsible disclosure, six additional cases remain under embargo and are withheld pending maintainer response

Bad-patched CVE	Bug Type	Project
CVE-2025-53010	NULL Pointer Dereference	MaterialX
CVE-2024-35230	Incorrect Control Flow	GeoServer
CVE-2023-27586	Server-Side Request Forgery (SSRF)	CairoSVG
CVE-2023-45897	Out-of-Bounds Read	exfatprogs
CVE-2023-33977	Improper Input Validation	Kiwi
CVE-2023-50266	Server-Side Request Forgery (SSRF)	Bazarr
CVE-2023-26145	Improper Input Validation	pydash

This pattern is consistent with the distinction between the two failure types. Incorrect patches often expose a localized realization error that can be corrected once identified. Incomplete patches, by contrast, require developers to discover what the initial patch failed to cover: the completion often depends on gradually expanding the fix to the full vulnerable surface, which takes longer even though not much more follow-ups.

Incomplete patches are harder to finish, taking notably longer and slightly more follow-up commits, than incorrect ones.

6.8 Discovery of Lingering Zero-Days

As noted earlier, the final patch (future commit) is typically the final resolving patch. However, using PatchAudit, we found 13 zero-day bad patches—i.e., even after all future commits, the latest main-branch snapshot of the (initially) bad-patched software remains vulnerable, as summarized in Table 7. We followed responsible disclosure for all these findings; 7 have been confirmed by respective developers, and one has been assigned a new CVE.

In the example of Figure 14 (CVE-2023-50266 in Bazarr), the initial patch applied a protocol check as `protocol.lower` instead of `protocol.lower()`, so the guard always passed. A follow-up commit corrected the typo but added no destination validation, leaving the SSRF surface still *exploitable in the latest snapshot*.

Root-Cause Patterns. We inspected the 13 zero-day bad patches and identified three patterns of causes of these lingering zero-days.

Interface-level policy inconsistency: a security policy is enforced on the primary interface but not applied to auxiliary endpoints or protocol handlers. In one case still under disclosure embargo, authentication is enforced in the web UI but a terminal WebSocket endpoint still accepts unauthenticated sessions.

Validation-coverage gaps: the fix validates the attack input partially but omits an equally critical dimension. In Bazarr (CVE-2023-50266), the scheme is restricted but the request target is not validated at all (Figure 14). In pydash (CVE-2023-26145), the restriction is enforced in `get()` but not in `invoke()`, enabling access via an alternate call path.

Missed equivalent instances: the developer patches one or more locations but fails to propagate the fix to parallel code sites. In MaterialX (CVE-2025-53010), multiple patches add NULL-guard checks to some iteration loops, yet `getShaderNodes()` still has an unguarded loop dereferencing a NULL node in latest snapshot.

Bad-patch auditing can surface live zero-days, not just explain past failures. PatchAudit uncovered 13 lingering (zero-day) bad patches (7 confirmed by maintainers) across 13 projects. Their root causes mirror the dominant pattern in the full bad-patch dataset: the intended security property was only partially propagated across interfaces, validation dimensions, or parallel code locations.

```
Commit message: prevent arbitrary file read and blind SSRF.
Commit ID: 17add7f
File Path: bazarr/app/ui.py
1 @ui_bp.route('/test/<protocol>/<path:url>', methods=['GET'])
2 def proxy(protocol, url):
3 + if protocol.lower not in ['http', 'https']:
4 +     return dict(status=False, error='Unsupported protocol')
5     url = protocol + '://' + unquote(url)
Commit message: fix test button bug introduced in previous commit
Commit ID: 84e8872
File Path: bazarr/app/ui.py
1 def proxy(protocol, url):
2 - if protocol.lower not in ['http', 'https']:
3 + if protocol.lower() not in ['http', 'https']:
4     return dict(status=False, error='Unsupported_protocol')
Commit message: Latest analyzed snapshot
Commit ID: 396a2d6
File Path: bazarr/app/ui.py
1 def proxy(protocol, url):
2     if protocol.lower() not in ['http', 'https']:
3         return dict(status=False, error='Unsupported_protocol')
4     url = f'{protocol}://{unquote(url)}'
5     result = requests.get(url, params, allow_redirects=False,
6         verify=False, timeout=5, headers=HEADERS)
```

Figure 14: A zero-day bad patch due to validation-coverage gaps. C_1 restricted the user-controlled scheme, C_2 corrected the buggy scheme check, but the later snapshot (396a2d6) still forwarded attacker-controlled HTTP(S) destinations to `requests.get()` without validating the request target.

7 Discussion

Operational Lessons. The failure causes measured in §6.3 indicate where patch review effort is best spent, and they point in different directions for the two failure modes. Two of the five cause categories in Table 5 are forms of under-scoped remediation: *Edge Case Missed* (31.8% of all bad patches) and *All Scenarios Not Considered* (31.6%), together 63.4%. Incomplete patches concentrate there most sharply—within that class, *Edge Case Missed* accounts for 35.7% and *All Scenarios Not Considered* for 37.0%—where the fix addresses the reported instance but not the full attack surface. One lesson is that vulnerability reports commonly center on a single triggering input, and that developers patch that instance without enumerating the class of inputs and paths sharing the same weakness. The `gpac` integer-overflow fix (§6.3) exemplifies this: it guarded `NbBits > 32` but missed the equally unsafe `NbBits == 0` boundary, leaving an equivalent path exploitable. Such patches call for “what did the patch fail to reach?” checks over boundary values, input variants, exceptional paths, equivalent locations, and interfaces, rather than confirmation that the reported case is fixed.

Incorrect patches instead concentrate in the remaining three categories, which capture failures of realization rather than of scope: within that class, *Incorrect Implementation* accounts for 41.9%, *Incorrect Conditioning* for 19.4%, and *Data Path Inconsistency* for 1.9%, or 63.2% in total. The two leading sub-patterns are *Logic Inconsistency* (18.8% of incorrect patches) and *Wrong Variable/Pointer Usage* (13.8%): the scope is understood, but the fix is encoded wrongly. Logic inconsistency is exemplified by the RELIC padding fix (§6.3), where the verification predicate was inverted, so the patch accepted the inputs it was meant to reject. Such defects leave the reported case apparently handled and escape scope-based review; they call instead for semantic validation of the introduced logic—predicates, API choices, ownership, and state propagation.

Compounding both, remediation is becoming harder to trace: among bad patches, follow-up commits that explicitly reference the original CVE or patch fell from 51.9% before 2020 to 23.3% since

(§6.2). Advisories that cite only the initial patch can make remediation appear complete even when the effective fix spans a longer, undocumented chain. Thus, the unit of remediation is the patch chain, not the initial patch—a chain that reviewers and downstream integrators often cannot recover from advisory metadata alone.

Practical Recommendations. Our study shows that bad CVE patches are common, often silently amended, and in some cases remain exploitable in current project snapshots. These results suggest several practical implications that go beyond generic secure-coding advice; PatchAudit can also support other applications.

Perspective 1: Patch validators and security-review tools. Our cause analysis suggests that patch validation should check the *obligations implied by the patch*, not only the edited lines. For an initial patch, a practical validator should expand from the changed code to the patch-relevant control/data-flow region and check three recurring obligations: (i) whether equivalent paths and code sites receive the same fix, (ii) whether the intended security policy is enforced consistently across related interfaces, and (iii) whether validation covers all security-critical input dimensions. The output need not be a binary verdict; it could report each obligation as *met*, *violated*, or *ambiguous*, reserving expensive follow-up—targeted tests, manual review, or AI-assisted analysis—for ambiguous cases.

Perspective 2: Vulnerability databases and advisory platforms. Our temporal analysis reveals a visibility gap in current remediation tracking: many corrective commits do not mention the original CVE or initial patch, so advisories that link only to initial patches can make remediation appear complete when the effective fix spans a longer patch chain. Vulnerability databases, package ecosystems, and advisory platforms could support *patch-chain metadata*: links from a CVE not only to the initial fix, but also to later commits that amend that fix. Such metadata would help downstream maintainers distinguish a merely patched version from a fully remediated version, reducing the risk of relying only on the faulty initial patch.

Perspective 3: Maintainers and downstream integrators. For security fixes, maintainers should treat follow-up commits that touch the same validation, authorization, memory-management, or state-propagation logic as potential continuation of the CVE fix, even when they lack CVE references. Downstream integrators should therefore audit the patch chain before backporting: applying only the first CVE-linked commit may preserve the vulnerable surface that later upstream commits silently corrected.

Perspective 4: Test suites and continuous integration. Our cause analysis suggests that most bad patches are under-scoped rather than wrong, so many of them are failures a suitable test suite could have caught. A fix that resolves the reported instance but overlooks its *equivalence class* would pass a single-case regression test yet leave the vulnerability reachable. For a security fix, developers should therefore add tests that exercise not only the reported input but the class it belongs to: (i) sibling boundary and extreme values, (ii) alternative input encodings and normalizations, and (iii) parallel entry points that share the same security-relevant behavior. Paired with the patch-induced obligations of Perspective 1, which identify *which* equivalent paths and input dimensions remain uncovered, such tests can turn an under-scoped fix into a build failure before release rather than a latent bad patch surfaced only by a later commit. For input-variation defects, property-based or fuzz testing over these dimensions can generate such cases automatically.

Limitations. Our current study has several limitations.

First, (1) PatchAudit identifies bad patches only through evidence of a later commit revisiting the same vulnerability, so deficient patches never revisited fall outside its scope by construction. We refer to these out-of-scope cases as *unmanifested* bad patches, which makes our 9.4% prevalence a manually validated lower bound over the observable class. To gauge this unobservable region, we manually audited a stratified random sample of 400 PatchAudit-negative CVEs—drawn proportionally across years and languages from the 5,918 (6,535 – 617) non-confirmed CVEs—checking each project’s latest main-branch snapshot for a still-exploitable initial patch with no future commit. We found 5 such cases, a negative-set miss rate of 1.25% (95% Wilson confidence interval: 0.54%–2.89%). We report this as a false-negative sensitivity check rather than a recall-corrected estimate of the true prevalence of faulty patches.

In addition, (2) we cover only CVEs whose advisories link to a GitHub-hosted patch in C/C++, Java, or Python, as our slicing backend supports these languages; other languages and non-GitHub sources fall outside our census. (3) Our static analysis also has inherent imprecision and limited recall, and our LLM reasoning may misinterpret rare idioms or atypical fix structures despite grounding checks; each pipeline stage may therefore reject some true cases. Moreover, (4) each bad patch receives a single top-level label; our analysis does not further quantify how the two failure modes may interact within a single remediation effort. Yet in general dual-mode (i.e., both incomplete and incorrect) cases may exist, making our single-label classification a limitation, although we did not observe those cases in our validated dataset. Finally, (5) responsible-disclosure obligations on still-live zero-day cases limit the exploit-level detail we can report.

8 Related Work

Empirical studies of patch quality. Prior studies show that security fixes can be incomplete, incorrect, or require follow-up changes [13, 31]. Others study vulnerability-contributing commits [18] or the broader patch lifecycle [1], including patch management, disclosure, propagation, and evolution [4, 15, 28, 34]. These works establish that patching is imperfect, but typically rely on manual inspection, process-level metrics, or explicit linkage signals. Our work conducts a large-scale, intent-aware measurement of manifestly faulty CVE patches and their lingering zero-day risk.

Identifying incorrect or incomplete fixes. Beyond security patches, prior work detects bad functional fixes using history-based signals, bug-neighborhood analysis, or supplementary patches [10, 26, 27, 37]. Patch-correctness assessment in automated program repair (APR) targets overfitting or plausible-but-wrong generated patches using semantic, learned, runtime, or LLM-based signals [12, 39]. These techniques mostly target generated or generic bug-fix patches and often assume tests, traces, or APR artifacts unavailable for many human-written CVE fixes.

Patch validation and security-intent analysis. Testing- and fuzzing-based validators check whether developer patches remove known bug behavior or avoid regressions using symbolic execution, directed fuzzing, or PoC-guided testing [8, 17, 22, 33]; however, they require executable environments, PoCs, and testing infrastructure that are difficult to obtain uniformly across thousands of historical CVE patches. Recent work also studies vulnerability-fix [14] or

silent-fix [38] identification, patch rationale generation [24, 35, 36], binary-level verification [32], patch propagation [16], and repair synthesis [11, 21, 23]. In contrast, we use patch-intent reasoning to identify follow-up commits that evidence faulty initial CVE fixes, enabling measurement of prevalence, failure causes, exposure windows, and lingering zero-day bad patches at scale.

9 Conclusion

We presented a large-scale measurement of bad security patches across 6,535 CVEs and 1,649 GitHub repositories. We found that 9.4% of initial CVE patches are manifestly faulty, mostly due to under-scoped remediation rather than incorrect implementation. These failures are also increasingly silent. To enable this study, we developed PatchAudit, an intent-oriented auditing pipeline evaluated as effective. Beyond retrospective measurement, we also uncovered 13 lingering zero-days, indicating major risks of bad patches.

Acknowledgment

We thank reviewers for their constructive comments. This work was supported by the National Science Foundation (NSF) under Grant CCF-2505223, 2551196, and 2623059, and Office of Naval Research (ONR) under Grant N000142212111 and N000142512252.

References

- [1] Kemal Altinkemer, Jackie Rees, and Sanjay Sridhar. 2008. Vulnerabilities and patches of open source software: an empirical study. *JISec* 4, 2 (2008), 3–25.
- [2] Juliet Corbin and Anselm Strauss. 2014. *Basics of qualitative research: techniques and procedures for developing grounded theory*. Sage publications.
- [3] Ericsson. 2023. Software vulnerability: impact & ways to avoid it. <https://www.ericsson.com/en/security/vulnerability-management>
- [4] Sadeq Farhang, Mehmet Bahadır Kırdan, Aron Laszka, and Jens Grossklags. 2019. Hey Google, what exactly do your security patches tell us? A large-scale empirical study on Android patched vulnerabilities. *arXiv:1905.09352* [cs.CR]
- [5] Forbes Technology Council. 2023. Zero-day vulnerabilities: 17 consequences and complications. <https://www.forbes.com/councils/forbestechcouncil/2023/05/26/zero-day-vulnerabilities-17-consequences-and-complications>
- [6] GitHub. 2025. GitHub security advisories. <https://github.com/advisories>
- [7] Ilan Peleg. 2023. The high cost of security vulnerabilities. <https://www.forbes.com/councils/forbesbusinesscouncil/2023/04/10/the-high-cost-of-security-vulnerabilities-why-observability-is-the-solution>
- [8] Zhiyuan Jiang, Shuitao Gan, Adrian Herrera, Flavio Toffalini, Lucio Romero, Chaojing Tang, Manuel Egele, Chao Zhang, and Mathias Payer. 2022. Evocatio: conjuring bug capabilities from a single PoC. In *CCS*. 1599–1613.
- [9] Joern Project. 2025. Joern: the bug hunter's workbench. <https://joern.io/>
- [10] Mijung Kim, Saurabh Sinha, Carsten Görg, Hina Shah, Mary Jean Harrold, and Mangala Gowri Nanda. 2010. Automated bug neighborhood analysis for identifying incomplete bug fixes. In *ICST*. 383–392.
- [11] Youngjoon Kim, Sunguk Shin, Hyoungshick Kim, and Jiwon Yoon. 2025. Logs in, patches out: automated vulnerability repair via tree-of-thought LLM analysis. In *USENIX Security*. 4401–4419.
- [12] Thanh Le-Cong, Duc-Minh Luong, Xuan Bach D. Le, David Lo, Nhat-Hoa Tran, Bui Quang-Huy, and Quyet-Thang Huynh. 2023. Invalidator: automated patch correctness assessment via semantic and syntactic reasoning. *TSE* 49, 6 (2023), 3411–3429.
- [13] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *CCS*. 2201–2215.
- [14] Wen Li, Li Li, and Haipeng Cai. 2022. On the vulnerability proneness of multilingual code. In *FSE*. 847–859.
- [15] Shuhan Liu, Jiayuan Zhou, Xing Hu, Filipe Roseiro Cogo, Xin Xia, and Xiaohu Yang. 2025. An empirical study on vulnerability disclosure management of open source software systems. *TOSEM* 34, 7, Article 214 (2025), 31 pages.
- [16] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. 2020. SPIDER: enabling fast patch propagation in related software repositories. In *S&P*. 1562–1579.
- [17] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: high-coverage testing of software patches. In *FSE*. 235–245.
- [18] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejada, Matthew Mokary, and Brian Spates. 2013. When a patch goes bad: exploring the properties of vulnerability-contributing commits. In *ESEM*. 65–74.
- [19] MITRE Corporation. 2025. CWE-1000: research concepts. <https://cwe.mitre.org/data/definitions/1000.html>
- [20] National Institute of Standards and Technology (NIST). 2025. National Vulnerability Database (NVD). <https://nvd.nist.gov>
- [21] Yu Nong and Haipeng Cai. 2026. DEMO: SemCRS: Streamlining vulnerability discovery and patching with LLMs. In *CCS*.
- [22] Yu Nong and Haipeng Cai. 2026. Neuro-symbolic proof-of-vulnerability generation with open-weight models. *arXiv preprint arXiv:2608.04217* (2026).
- [23] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2025. AP-PATCH: automated adaptive prompting large language models for real-world software vulnerability patching. In *USENIX Security*. 4481–4500.
- [24] Yu Nong, Guangbei Yi, Mohammed Aldeen, Long Cheng, Hongxin Hu, and Haipeng Cai. 2026. Assessing and improving prompting large language models for software vulnerability analysis. *TOSEM* (2026).
- [25] Offensive Security. 2025. Exploit database. <https://www.exploit-db.com/>
- [26] Jihun Park, Miryung Kim, and Doo-Hwan Bae. 2017. An empirical study of supplementary patches in open source projects. *EMSE* 22, 1 (2017), 436–473.
- [27] Jihun Park, Miryung Kim, Baishakhi Ray, and Doo-Hwan Bae. 2012. An empirical study of supplementary bug fixes. In *MSR*. 40–49.
- [28] Guido Schryen. 2009. A comprehensive and comparative analysis of the patching behavior of open source and closed source software vendors. In *IMF*. 153–168.
- [29] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? Overfitting in automated program repair. In *FSE*. 532–543.
- [30] Haoye Tian, Kunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F. Bissyandé. 2022. Is this change the answer to that problem?: correlating descriptions of bug and code changes for evaluating patch correctness. In *ASE*. 1–13.
- [31] Seunghoon Woo, Eunjin Choi, and Heejo Lee. 2025. A large-scale analysis of the effectiveness of publicly reported security patches. *COSE* 148 (2025), 104181.
- [32] Hongwei Wu, Jianliang Wu, Ruoyu Wu, Ayushi Sharma, Aravind Machiry, and Antonio Bianchi. 2025. VeriBin: adaptive verification of patches at the binary level. In *NDSS*. 1–18.
- [33] Yuhang Wu, Zhenpeng Lin, Yueqi Chen, Dang K. Le, Dongliang Mu, and Xinyu Xing. 2023. Mitigating security risks in Linux with KLAUS: a method for evaluating patch correctness. In *USENIX Security*. 4247–4264.
- [34] Zifan Xie, Ming Wen, Zichao Wei, and Hai Jin. 2024. Unveiling the characteristics and impact of security patch evolution. In *ASE*. 1094–1106.
- [35] Xu Yang, Wenhan Zhu, Michael Pacheco, Jiayuan Zhou, Shaowei Wang, Xing Hu, and Kui Liu. 2025. Code change intention, development artifact, and history vulnerability: putting them together for vulnerability fix detection by LLM. *PACMSE* 2, FSE (2025), 489–510.
- [36] Guangbei Yi, Yu Nong, Minzhang Li, and Haipeng Cai. 2026. Exploring and improving real-world vulnerability data generation via prompting large language models. In *ICSE*. 2044–2056.
- [37] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How do fixes become bugs?. In *FSE*. 26–36.
- [38] Jiayuan Zhou, Michael Pacheco, Jinfu Chen, Xing Hu, Xin Xia, David Lo, and Ahmed E. Hassan. 2023. CoLeFunDa: explainable silent vulnerability fix identification. In *ICSE*. 2565–2577.
- [39] Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Hung Huu Nguyen, Thanh Le-Cong, Junda He, Bach Le, and David Lo. 2024. Leveraging large language model for automatic patch correctness assessment. *TSE* 50, 11 (2024), 2865–2883.

Ethical Considerations

We use only public repositories, public CVE records, and off-the-shelf static-analysis outputs; we do not access private systems or user data. For the 13 exploitable zero-day bad patches, we used PoCs only to confirm impact, privately reported them to maintainers, and embargo exploit-enabling details. Unresolved cases are anonymized or reported only in aggregate/high-level form when project identity plus bug type could increase risk.

Open Science

Our complete artifact, including the PatchAudit source code, prompts/reasoning schemas, fine-tuning data/scripts/weights, BadPatchBench labels and construction scripts, evaluation outputs, analysis scripts, and the 617-CVE bad-patch dataset, is available at <https://doi.org/10.5281/zenodo.22698135>. PatchAudit is also actively maintained at <https://github.com/SRestLabUB/PatchAudit>.