ARTIFACT EVALUATED
IEEE S&P
AVAILABLE

# SoK: From Finding to Deployment: Systematizing the OS Kernel Bug Lifecycle

Luyao Bai*, Gengda She†, Kenan Alghythee*, Hang Zhang†, and Xiaoguang Wang*
*University of Illinois Chicago †Indiana University Bloomington

***Abstract*—Automated kernel bug discovery has advanced rapidly. Continuous fuzzing and static analysis systems, such as syzbot, now expose Linux kernel bugs at a scale that downstream processes struggle to absorb. Yet a crash report is only the beginning. Before a bug is eliminated, it must be triaged, understood, patched, validated, reviewed, integrated, and often backported. These later stages remain far less automated, creating a persistent gap between bug discovery and patch deployment.**

**This SoK systematizes the Linux kernel bug lifecycle from discovery to deployment. We organize prior work and production systems into five stages: discovery, triage, patch generation, patch validation, and integration. We explain the resulting automation gradient through kernel-specific challenges such as concurrency, implicit invariants, cross-syscall state, hardware dependence, lack of fault isolation, and architecture/configuration multiplicity. We further ground the analysis in a measurement of real syzbot-fixed bugs. The data shows that the crash-to-patch gap is not merely a backlog of unfixed reports but a structural failure mode of the repair pipeline: even after being fixed, bugs often remain open for weeks, require review-driven patch revisions, or lack reproducers that current repair and validation systems assume. This exposes a mismatch between where kernel-security automation is mature and where bug closure actually breaks down. These findings expose a deeper mismatch: today's repair and validation techniques often assume reliable reproducers, localized root causes, and checkable correctness oracles, yet these are precisely the artifacts missing from many real kernel bug reports. Closing the crash-to-patch gap, therefore, requires treating such artifacts as outputs to be produced, not prerequisites to be assumed.**

***Index Terms*—Operating system security, Linux kernel, Bug lifecycle, Fuzzing, Automated program repair, Patch validation, Systematization of knowledge**

## 1. Introduction

The operating system (OS) kernel is the largest and most privileged trusted computing base on virtually every device. The Linux kernel alone exceeds 30 million lines of code, integrates thousands of patches per release, and is maintained by a loosely coordinated community of volunteers and corporate contributors. A single memory-safety or concurrency defect can compromise the entire system, and memory-safety errors continue to account for roughly 70% of serious vulnerabilities in large C/C++ codebases including the kernel [1]. Securing the kernel is therefore not a single problem but a pipeline of problems: a bug must be found, understood and triaged, repaired and validated, and finally reviewed, integrated into a constantly moving codebase, and backported to the stable trees.

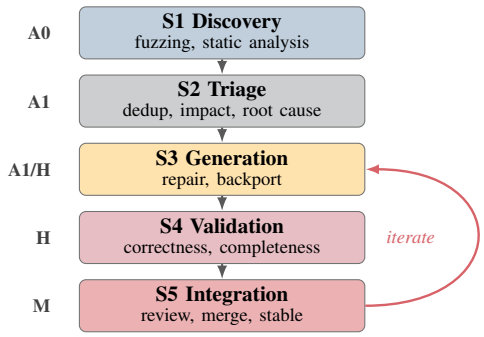


Figure 1. The OS kernel bug lifecycle in five stages, from DISCOVERY to INTEGRATION, with the automation level of each stage (A0 to M).

The first stage of this pipeline has been transformed. Coverage-guided kernel fuzzing now runs continuously at scale: Google's syzbot fuzzes mainline and linux-next around the clock and has reported tens of thousands of bugs, and static analysis has scaled to the whole kernel. Continuous fuzzing and static analysis now report candidate bugs faster than the downstream pipeline can process and close them; the bottleneck is no longer finding more bugs but clearing the backlog of bugs already found.

The remaining stages have not scaled with it. Everything after discovery is still predominantly manual work for an overworked maintainer population [2], [3]: a maintainer must triage a crash into an actionable root cause, judge its security impact, write a patch preserving kernel invariants, show that it neither regresses nor partially fixes the bug, and shepherd it through mailing-list review and stable backporting. The visible symptom is a growing backlog, with fix latency varying enormously across subsystems. We refer to the distance between an automatically discovered crash and a deployed, validated fix as the crash-to-patch gap.

Most recently, large language models (LLMs) have been applied across the pipeline in four distinct roles: as artifact generators (syscall specifications, static checkers, candidate

patches, review comments), as classifiers/judges (severity, patch correctness), as agents that drive a repair or integration workflow, and, aspirationally, as reasoning engines for root cause and fix completeness. Yet this adoption is uneven and largely unsystematized, with strong evidence for the first two roles and thin evidence for the last.

Existing systematizations address slices of this pipeline: kernel-fuzzing surveys [4] cover discovery, the SoK on automated vulnerability repair [5] covers user-space repair, and the SoK on kernel hardening [6] covers exploitation and mitigation, orthogonal to bug management. None unifies discovery with triage, repair, validation, and the socio-technical integration process, and none treats the kernel's defining characteristic, a pipeline automated at the front and manual at the back. We argue this end-to-end, kernel-specific view is exactly what is needed to direct the field's next decade of effort.

Overall, this paper makes the following contributions:

- We define the *OS kernel bug lifecycle* as a five-stage pipeline and use it to systematize 140 research efforts and production systems against a common set of dimensions (Section 3–9).
- We articulate the *automation gradient* as the field's defining structural property and tie it to *six kernel-specific challenges* (Section 4) whose difficulty is back-loaded onto the later stages, explaining where LLMs have and have not closed the gap.
- We conduct a *longitudinal measurement of the crash-to-patch gap* over 6,946 fixed syzbot bugs, decomposing fix latency into pipeline segments, locating where the delay sits, and computing a repair-readiness score (Section 10).
- We build a *coverage-gap matrix* mapping existing techniques onto (bug class $\times$ lifecycle stage), exposing combinations no current tool addresses, and distill *takeaways* and *open problems* per stage.
- We release our dataset, classification, and analysis scripts as public artifacts[1] to support reproducibility and continued community curation.

We focus on the kernel and the management of bugs from discovery to deployment, treating user-space techniques (general APR, code-review research) as contrast to highlight what is genuinely kernel-specific. Exploitation and runtime hardening are out of scope, as they concern defending against bugs rather than fixing them and are covered by a complementary SoK [6].

# 2. Methodology and Scope

## 2.1. Paper Selection

We assembled our corpus in three steps. *Seed search*: we queried DBLP, Google Scholar, and the proceedings of top security (S&P, USENIX Security, CCS, NDSS), systems (OSDI, SOSP, EuroSys, ATC, ASPLOS), and software-engineering (ICSE, FSE, ASE, ISSTA) venues, combining kernel with bug discovery, vulnerability repair, patch generation, patch correctness, backporting, and code review, over a primary window of 2015–2026 (the rise of continuous kernel fuzzing through the LLM era), admitting seminal earlier work such as the *Faults in Linux* studies [145], [146] where it anchors a category. *Filtering*: we retained a paper if it (i) targets or substantially evaluates on the OS kernel and (ii) contributes to at least one lifecycle stage, admitting a bounded set of user-space and software-engineering papers as contrast where they expose a kernel-specific gap, and excluding work targeting solely exploitation or runtime hardening. *Snowballing*: we chased citations forward and backward until no new methodologically distinct work appeared (two iterations). The final corpus totals 140 papers.

## 2.2. The Lifecycle Lens

We organize the corpus around the five lifecycle stages a kernel bug traverses from existence to eradication (Figure 1): S1 DISCOVERY exposes a latent defect as an observable failure (Section 5); S2 TRIAGE & UNDERSTANDING deduplicates, root-causes, and assesses impact (Section 6); S3 PATCH GENERATION synthesizes a candidate fix, including backports (Section 7); S4 PATCH VALIDATION establishes that the fix resolves the bug, preserves functionality, and is complete (Section 8); and S5 INTEGRATION reviews, merges, and ships it through the community process (Section 9). Three kinds of callout boxes thread the paper: Takeaways synthesize each stage, Open Problems mark unresolved challenges, and Findings report our measurement results (Section 10).

## 2.3. The Automation Gradient

Our unifying claim is structural. We assign each surveyed technique an automation level on a four-point scale that distinguishes deployed from merely demonstrable automation:

- **A0**: fully automatic in a production pipeline with no human in the loop (e.g., syzbot's continuous fuzzing).
- **A1**: fully automatic per input, but offline, per-bug, or a prototype outside any continuous pipeline (e.g., current LLM repair agents).
- **H**: human-guided, where the tool proposes and a human decides.
- **M**: manual best practice in which a tool merely assists (e.g., mailing-list code review).

The distinction matters because much back-end "automation" is A1: it works in a paper but has never been wired into the syzbot-scale flow, so it does not relieve the production bottleneck. The modal level degrades monotonically across S1$\rightarrow$S5, and only discovery reaches A0. Section 3 makes this gradient precise, the per-stage sections substantiate it, and Section 10 shows its consequence in the wild as the crash-to-patch gap.

1. https://sysec-uic.github.io/sok-kernel-bug-lifecycle/

| | |
|---|---|
| **C1** Concurrency | Razzer [7] SegFuzz [8] Snowboard [9] DCUAF [10] Double-Fetch [11] KRACE [12] Snowcat [13] UACatcher [14] DEADLINE [15] |
| **C2** Implicit invariants | Hydra [16] CountDown [17] DR.CHECKER [18] K-Miner [19] CRIX [20] LRSan [21] UBITect [22] K-MELD [23] Goshawk [24] STACK [25] KNighter ★ [26] Coccinelle [27] PATA [28] KUBO [29] LLift ★ [30] IMMI [31] MANTA [32] CRED [33] CID [34] PeX [35] CheQ [36] Err-Spec [37] DEPA [38] Uninit-Bin [39] kGym [40] CrashFixer ★ [41] PatchIsland ★ [42] Beyond-C2P [43] ChatRepair ★ (U) [44] Learn2Fix (U) [45] RGym ★ [46] RepairAgent ★ (U) [47] AutoCodeRover ★ (U) [48] ThinkRepair ★ (U) [49] KLAUS [50] Patch-Impact (U) [51] LLM-judge ★ (U) [52] Incomplete-Fixes [53] PVBench (U) [54] LiveBench [55] |
| **C3** Cross-call state | kAFL [56] Horus [57] BoKASAN [58] MoonShine [59] HEALER [60] MOCK [61] ACTOR [62] SyzVegas [63] Snowplow [64] IMF [65] SyzGen [66] KSG [67] SyzDescribe [68] KernelGPT ★ [69] JANUS [70] Hydra [16] StateFuzz [71] HFL [72] SyzDirect [73] syzkaller [74] Bin-Cov [75] SyzGen++ [76] FuzzNG [77] SUTURE [78] BugLens ★ [79] UAFX [80] Dup-reports [81] SyzRetrospector [82] AURORA [83] ARCUS [84] Igor [85] |
| **C4** Hardware | Unicorefuzz [86] Agamotto [87] SyzDescribe [68] DIFUZE [88] DR.FUZZ [89] PrIntFuzz [90] KextFuzz [91] NTFUZZ [92] DR.CHECKER [18] Nyx [93] USBFuzz [94] ReUSB [95] Fast-fixes [96] |
| **C5** No isolation | Digtool [97] SCAD [98] SyzScope [99] DiffCVSS [100] LLM-triage ★ [101] Vuln-Prediction [102] GREBE [103] KOOBE [104] K-LEAK [105] FUZE [106] KEPLER [107] |
| **C6** Arch/multi-tree | SyzRisk [108] Coccinelle [27] IncreLux [109] Kconfig [110] DiffCVSS [100] PatchScout [111] DisPatch [112] SPAIN [113] SyzBridge [114] SPI [115] Collateral-Evol. [116] FixMorph [117] PatchNet [118] PatchScope [119] Patch-porting [120] PDiff [121] PS3 [122] Patch-Me-If-Can [123] CVE-coord. [124] Seamless-upd. [125] |
| **Socio-technical** | Tufano'21 ★ (U) [126] CodeReviewer ★ (U) [127] AUGER (U) [128] ReviewBench ★ (U) [129] DPO-f+ ★ (U) [130] ReviewStudy ★ (U) [131] Bacchelli (U) [132] McIntosh (U) [133] Ruangwan (U) [134] Goncalves (U) [135] Incivility [136] Patch-comm. [137] Zhou [2] Tan [3] Onboarding (U) [138] Rust-for-Linux [139] Women-in-OSS (U) [140] Jiang [141] Disclosure (U) [142] Coverity-alerts (U) [143] Patchwork [144] |

**Pill color = lifecycle stage:** S1 Discovery S2 Triage S3 Generation S4 Validation S5 Integration. ★ = LLM-based (U) = user-space contrast. A paper that confronts several challenges appears in several rows.

Figure 2. Classification of the surveyed systems by kernel challenge (rows, C1–C6) and lifecycle stage (pill color, S1–S5).

## 3. The Kernel Bug Lifecycle at a Glance

Figure 1 presents the five-stage lifecycle and the automation gradient that is this paper's organizing thesis. We call the early, automated stages (S1–S2) the front end and the later, human-in-the-loop stages (S3–S5) the back end: reading top to bottom, production-deployed automation (A0) thins out, only discovery reaches it, and the later stages lean on offline prototypes (A1) and human judgment (H, M).

Automation level is only one dimension per stage. Table 1 broadens this into the full framework we use throughout, recording for each stage the artifacts it consumes and produces, its dominant method, automation level, the oracle that defines when the stage is "done", the role LLMs play, and the challenges (Section 4) that constrain it. Reading top to bottom, every column weakens together; the gradient is this same decline, seen along five axes at once.

DISCOVERY runs unattended and produces bugs faster than they can be processed; TRIAGE automates deduplication and impact but not root cause; GENERATION proposes patches at low accepted yields; VALIDATION is largely manual; and at INTEGRATION the limit is maintainer bandwidth itself. This gradient is not an accident of effort allocation; Section 4 argues it follows from six kernel-specific properties whose difficulty falls most heavily on exactly these back-end stages.

## 4. Why Kernel Bugs Are Different

A natural objection to a kernel bug-lifecycle SoK is that it merely re-targets user-space bug finding and automated program repair (APR). This section answers that objection and supplies the analytical lens for the rest of the paper: six cross-cutting properties of the OS kernel that shape every stage of its bug lifecycle (Section 4.1), each grounded in a representative merged fix from our corpus (Section 4.2). These properties burden the pipeline asymmetrically, falling far more heavily on the later stages than on discovery, which aligns with and helps explain the automation gradient (Section 4.3).

### 4.1. Six Cross-Cutting Challenges

**C1: Pervasive concurrency and weak memory ordering.** The kernel executes concurrently on all CPUs, with preemption, interrupts, and RCU. Many defects (data races, deadlocks, use-after-free via concurrent free) are properties of a particular interleaving, not of an input, whereas most user-space APR and fuzzing assume sequential, input-determined behavior. An entire kernel sub-field exists just to control interleavings (Section 5).

**C2: Implicit, unspecified invariants.** Kernel correctness rests on conventions no machine-checkable artifact records: lock-ordering discipline, reference-count balance, RCU grace periods, the ban on sleeping in atomic context, object-ownership rules. No test suite encodes them. This is the deepest difference from user-space APR, whose generate-and-validate loop relies on tests as a proxy for the specification.

**C3: Cross-syscall, long-lived state.** Kernel objects persist across system calls, so triggering a bug requires a precise

TABLE 1. A CROSS-CUTTING TAXONOMY OF THE KERNEL BUG LIFECYCLE, APPLIED UNIFORMLY ACROSS ALL FIVE STAGES.

| Stage | Consumes → Produces | Dominant method | Auto. | Oracle ("done") | Productive LLM role | Chal. |
|---|---|---|---|---|---|---|
| S1 Discovery | source/binary → crash + report | coverage-guided search, static analysis | A0 | *strong*: crash / sanitizer trips | generator: specs, checkers | C1,C3,C4 |
| S2 Triage | crash + report → root cause, severity, dedup | symbolic exec., static, learning | A1 | *partial*: exploit primitive, dup match | judge: severity labels | C3,C5 |
| S3 Generation | root cause → candidate patch | templates, transforms, LLM agents | A1/H | *weak*: a single reproducer | agent: propose patch | C2,C6 |
| S4 Validation | patch → correct/complete verdict | directed fuzzing, static, PoC | H | *weak/none*: no spec for completeness | judge: correctness (unverified) | C2,C5 |
| S5 Integration | patch → merged + backported fix | human review, social process | M | *social*: maintainer acceptance | generator: review comments | C6 |

sequence that drives the kernel into a particular state, and the observable symptom can be far removed from the offending instruction. User-space targets are frequently single-input, their crashes closer to their causes.

**C4: Hardware and peripheral dependence.** Device drivers constitute the majority of kernel code and depend on physical devices, memory-mapped I/O, DMA, interrupts, and firmware. Exercising or fixing them may require hardware unavailable in many test environments.

**C5: No fault isolation, whole-system blast radius.** The kernel has no process boundary to contain a fault: a single bug can corrupt arbitrary system state, failures can be silent, and a benign-looking `WARNING` may conceal an arbitrary write. A user-space crash is contained and cheap to roll back; a faulty kernel patch can render the system unbootable.

**C6: Architecture/configuration multiplicity and multi-tree deployment.** One kernel source compiles to many architectures and thousands of configuration options, and ships through mainline plus numerous stable and vendor trees. A fix must hold across that space and be propagated to every affected tree, an entire class of work (backporting, patch-presence testing) with no user-space analog.

## 4.2. The Challenges in the Wild

The six properties above are not abstractions. We mined the syzbot-fixed corpus of Section 10 for each property's footprint, tagging every fix by lexical and structural signals in its diff, commit message, and review threads,[2] and present one representative merged patch per challenge; for space, we include in-paper code examples only for C2, C3, and C6.

**C1, a lock-ordering fix in `io_uring`.** The normal I/O path acquires `uring_lock` then the `seq_file` lock; the `/proc` fdinfo path acquires them in the opposite order. The fix must reason about the global lock order rather than any single path, breaking the cycle with a `trylock`. In our corpus, 10.9% of all fixes edit a locking or memory-ordering primitive, and concurrency-class bugs lack any reproducer 44.7% of the time versus 22.4% for the rest.

**C2, a reference-count fix that repairs another fix. (Figure 3)** An earlier syzbot fix added an unconditional `llc_sap_hold/put` pair to keep a SAP alive across `release_sock()`, thereby violating a different implicit invariant: a `SOCK_ZAPPED` socket has no SAP at all. The follow-up patch (tagged `Fixes:` the first one) restores an object-lifetime rule no test suite encodes.

```
- sap = llc->sap;
- llc_sap_hold(sap);
- if (!sock_flag(sk, SOCK_ZAPPED))
+ if (!sock_flag(sk, SOCK_ZAPPED)) {
+ struct llc_sap *sap = llc->sap;
+ llc_sap_hold(sap);
llc_sap_remove_socket(llc->sap, sk);
- release_sock(sk);
- llc_sap_put(sap);
+ release_sock(sk);
+ llc_sap_put(sap);
+ } else {
+ release_sock(sk);
+ }
```

An earlier fix held a SAP that a **`SOCK_ZAPPED`** socket never owns. This **Fixes:** patch takes the refcount only when a SAP exists, restoring a lifetime invariant no test encodes (C2).

Figure 3. C2 in the wild: a reference-count fix in `net/llc` that repairs an earlier fix.

Such fix-of-a-fix chains are measurable: at least 4.1% of corpus fixes repair another fix in the corpus, and at least 2.1% were themselves later repaired again, a direct lower bound on invariant-violating "complete" patches.

**C3, a crash far from its cause. (Figure 4)** A general-protection fault manifested in VFS mount-parameter parsing, but the defect lived in the LSM layer, where stacked security modules disagreed about a hook's return-value contract. The fix rewrites the hook dispatcher in `security/security.c`, two subsystems away from the crash site.

In the corpus, 10.3% of fixes land outside the crashing directory and 4.2% land in a different subsystem entirely, and those displaced bugs take a median 55 days to fix versus 33 for the rest.

**C4, a driver fix validated only by emulation.** A managed-buffer leak in ALSA PCM hid on the release path that calls

2. Percentages in this subsection are computed over the same 6,946-bug corpus of Section 10; the per-challenge taggers are released with our artifact.

```
- return call_int_hook(fs_context_parse_param,
- -ENOPARAM, fc, param);
+ int rc = -ENOPARAM;
+ hlist_for_each_entry(hp, &heads.parse, list) {
+ trc = hp->hook.parse_param(fc, param);
+ if (trc == 0) rc = 0; // LSM claimed it
+ else if (trc != -ENOPARAM) return trc;
+ }
+ return rc;
```

A GPF in VFS mount parsing actually came from the **LSM hook's** return-value contract two subsystems away. The fix rewrites the dispatcher and SELinux's return value (C3).

Figure 4. C3 in the wild: a crash far from its cause, fixed in `security/security.c`.

```
#ifdef CONFIG_IEEE802154_NL802154_EXPERIMENTAL
+ if (wpan_dev->iftype == NL802154_IFTYPE_MONITOR)
+ goto out; // llsec mib not initialized
if (nl802154_get_llsec_params(...) < 0)
goto nla_put_failure;
#endif
```

The entire fix sits inside **`CONFIG_..._EXPERIMENTAL`**, then was backported to **seven stable trees** (4.4–5.11). Three lines of code, but a wide deployment fan-out (C6).

Figure 5. C6 in the wild: a config-conditional ieee802154 fix backported to seven stable trees.

the driver's `hw_free` callback directly. The fix factors the callback handling into one helper invoked from both paths. The defect lives behind a device-operations interface, and like the 16.1% of corpus fixes that touch a driver or sound path, its validation rests on syzbot's emulated devices rather than the physical hardware it abstracts.

**C5, a benign warning concealing a bounds bug.** syzbot reported only a `WARNING` in netlink's extended-ack path. The fix reveals the substance, as the bounds check for the attribute pointers compared against the wrong buffer, so the offset written back to user space could be computed from an address outside the message payload. About a quarter (26.5%) of corpus reports carry a benign-looking symptom class (`WARNING`, hang, stall), and for 10.8% of those the merged fix edits memory-safety-relevant code, the SyzScope risk-inversion at corpus scale.

**C6, a config-conditional fix that shipped to seven trees. (Figure 5)** An ieee802154 crash existed only under `CONFIG_IEEE802154_NL802154_EXPERIMENTAL`, and the entire fix sits inside that guard; the patch was then carried into seven stable trees (4.4 through 5.11). The fix itself is three lines, and the C6 burden is the deployment fan-out around it. Tagging only on unambiguous evidence (explicit `Cc: stable`, config-conditional code, or an `arch/` file), 17.2% of corpus fixes carry a C6 footprint, a lower bound: a further 2,329 fixes appear in stable backport threads (median four trees each) without an explicit tag.

### 4.3. The Burden Is Asymmetric

Figure 2 regroups the surveyed systems along both axes at once, the lifecycle stage each addresses and the challenge it confronts. The challenges concentrate toward the back of the pipeline, and the kind of difficulty differs by stage. At discovery, even high-burden properties are triggering difficulties that search can amortize away (perturbing schedules for C1, emulating devices for C4, inferring syscall dependencies for C3): one only needs to provoke the property once, and continuous fuzzing has unbounded attempts. The later stages face reasoning difficulties that search cannot dissolve: preserving the global locking discipline (C1), respecting invariants written down nowhere (C2), ruling out sibling instances across all configurations (C2, C6), or confirming a driver fix without the device (C4), while a wrong answer can corrupt the whole system (C5). These tasks lack exactly what would make them automatable, a specification, a test oracle, executable hardware, the safety of isolation; user-space APR matured because it has all four. This asymmetry is, we argue, the structural reason the gradient exists, and our measurement (Section 10) shows the cost is highest for exactly the bug classes (concurrency, use-after-free) whose challenges (C1, C2) are hardest to reason about.

## 5. S1: Bug Discovery

Discovery is the stage at which a latent defect is exposed as an observable failure (a crash, sanitizer report, or analyzer warning). It is the most thoroughly automated stage of the lifecycle and the reason the rest of the pipeline is under pressure: continuous fuzzing and whole-kernel static analysis produce candidate bugs faster than downstream stages can absorb them. As discovery is well served by existing surveys [4], we keep this section compact, covering dynamic (Section 5.1) and static (Section 5.2) discovery and the learning-augmented turn (Section 5.3); we tabulate representative systems per stage and the full S1 classification, including each system's technique families, in the appendix.

### 5.1. Dynamic Discovery: Kernel Fuzzing

Coverage-guided fuzzing is the dominant kernel bug-finding technique, anchored in practice by SYZKALLER and its continuous-integration front end syzbot [74], [147]. A kernel fuzzer must solve three problems that distinguish it from user-space fuzzing: obtain coverage feedback from privileged code, generate structured sequences of interdependent system calls, and reach deep states guarded by complex preconditions. The literature maps cleanly onto these problems.

**Coverage feedback and execution.** KAFL established that hardware-assisted tracing with a thin hypervisor yields general, low-overhead coverage even for closed-source kernels [56]; a body of follow-on work drives down execution

and instrumentation cost through emulation, VM checkpointing, snapshotting, binary-only sanitization, and richer feedback signals [86], [87], [57], [93], [58], [75]. With feedback largely commoditized, the field's attention shifted to input structure and state.

**Syscall structure and dependencies.** Because kernel state is built across syscall sequences, much of the field improves how sequences are constructed, by distilling seeds from traced syscall logs (MOONSHINE), learning inter-syscall influence and dependency relations, or casting mutation and scheduling as learning problems [59], [60], [61], [62], [63], [64]. A complementary line confronts the specification bottleneck: syzkaller's effectiveness depends on hand-written syscall descriptions, so a series of systems infers interface models for closed-source kernels or generate descriptions automatically from the kernel–driver contract [65], [66], [67], [68], [76], while FUZZNG sidesteps descriptions entirely by reshaping the input space around file descriptors and user pointers [77].

**Drivers and peripherals.** Driver code is vast, hardware-dependent (challenge C4), and a disproportionate source of bugs, motivating fuzzers that decouple drivers from physical devices, by reconstructing ioctl interfaces (DIFUZE), synthesizing or simulating fake device inputs, and extending interface-aware fuzzing to macOS and Windows kernels [88], [89], [90], [91], [92]. The USB stack, a large remote attack surface, is reached by device emulation and replay-guided fuzzing [94], [95].

**File systems and concurrency.** Stateful subsystems need domain-specific input models: JANUS and HYDRA jointly mutate file-system images and operations [70], [16]. Concurrency bugs require controlling interleavings (challenge C1), not just inputs. RAZZER pairs static race candidates with deterministic scheduling [7], and successors explore interleaving segments, inter-thread communication, data-race fuzzing for file systems, and learned guidance [8], [9], [12], [13]; statically, DCUAF mines concurrent use-after-free from lock patterns [10].

**Reaching deep state.** Coverage plateaus because many unreached branches depend on hard-to-synthesize kernel state [148]. Responses track state variables, add symbolic execution for guarded branches, or follow reference-count state [71], [72], [17]. A directed strand focuses scarce fuzzing budget on suspect code, target sites, and risky recent changes [73], [108], [149]. Several papers document the continuous setting directly [150], [151], which we revisit as evidence for the crash-to-patch gap (Section 10).

## 5.2. Static Discovery: Whole-Kernel Analysis

Static analysis trades soundness and false positives for the ability to reason about code paths fuzzing rarely reaches and to target specific bug classes. DR. CHECKER pioneered a "soundy" driver analysis [18], and K-MINER partitioned whole-kernel analysis per syscall [19]. A productive line infers implicit security rules from the kernel itself, flagging missing checks, lacking-recheck bugs, use-before-initialization, and double-fetch windows [20], [21], [22], [11]; others model a single defect family, such as object-ownership leaks or custom-allocator memory corruption [23], [24]. COCCINELLE occupies a special place: its semantic patches both find pattern bugs and fix them at scale, foreshadowing S3 [27]. Further lines reach the compiler and binary layers, catching unstable code discarded under undefined behavior [25] and memory bugs in binary-only kernels [97].

The defining tension is precision against scale. Path-sensitive typestate analysis, on-demand SMT-checked path constraints, incremental analysis across revisions, and cross-entry taint chaining all sharpen whole-kernel precision [28], [29], [109], [78]. Because the imprecise first stage can emit tens of thousands of candidates, recent work pairs these pipelines with LLMs to prune false alarms [30], [79]. A second thread targets bugs that span entry points and object lifetimes, connecting a free in one syscall to a use in another, racing device cleanup against concurrent syscalls, and checking allocation intention, accounting, reference-count consistency, and permission propagation [80], [14], [31], [32], [33], [34], [35].

Where no specification is written down, the analysis recovers one, from the kernel's own security checks, error-handling structure, historical fixes, or even the Kconfig option space [36], [37], [38], [110]. Representation choices matter too, from code property graphs to formalized double-fetch conditions, binary-level recovery, learned features, and static reasoning about network side channels [152], [15], [39], [153], [98].

## 5.3. The Learning-Augmented Turn

LLMs first entered the lifecycle at discovery, and most maturely at its specification bottleneck: KERNELGPT synthesizes syscall descriptions that previously required experts [69], and KNIGHTER synthesizes checkers, rather than findings, transferring analyst intent into reusable analyses [26]. The pattern is telling. At S1, LLMs automate expertise, not an already automated step.

**Takeaway 1**

Discovery is no longer the dominant bottleneck. It is not "solved" (hardware modeling, semantic bugs, and interleaving control remain open), but coverage feedback is commoditized, and the live frontiers raise the rate of an already-overflowing pipeline. LLMs here scale human expertise (specifications, checkers) into automation rather than displacing it.

**Open Problem 1**

Discovery is optimized in isolation from the downstream pipeline: fuzzers and analyzers are evaluated on bugs found, not bugs fixed. No discovery technique we surveyed prioritizes findings by downstream fixability, patch availability, or maintainer load, even though the binding constraint has moved downstream.

## 6. S2: Triage and Understanding

Discovery produces raw failures, and triage turns a failure into something a developer can act on: a syzbot crash must be deduplicated against known reports, its impact, severity, and exploitability assessed, its root cause identified, and, when a fix already exists upstream, its fixing commit located. This is where the automation gradient first bends. Several triage subtasks are automated, but the central one, root-cause analysis, remains expert-driven and is the practical throttle on everything downstream; we tabulate representative systems for this stage in the appendix (Section A).

**Deduplication.** At syzbot scale, the same defect surfaces under many distinct crash signatures, inflating the apparent bug count. Mu et al. [81] performed the defining study of duplicated kernel bug reports, showing that naive title/stack-trace bucketing over-merges and under-merges; SYZRETROSPECTOR attacks the same identity problem from the provenance side [82], and IGOR clusters crashes on root cause rather than surface signature [85]. Deduplication is automated but imperfect, and its errors propagate: a mis-merged report hides a distinct bug, while an over-split one wastes triage effort.

**Impact and severity.** Not all crashes deserve equal attention, and a fuzzer's reported symptom often understates the true risk. SYZSCOPE [99] showed that a large fraction of bugs syzbot labels "low-risk" in fact harbor high-risk primitives such as control-flow hijack, by symbolically exploring the states reachable from the crash. Related work recomputes severity per derived kernel version (DIFFCVSS), applies LLMs to streamline CVE/CVSS labeling, and predicts where risk concentrates with metric and text-mining models [100], [101], [102].

**Exploitability as a triage signal.** A bug's exploitation potential is a strong prioritization signal, and a line of work estimates it automatically, by exploring the alternative error behaviors a bug can manifest, extracting the capabilities of out-of-bounds writes, reasoning about leak chains and use-after-free exploitation, evaluating control-flow-hijack primitives, and testing whether an upstream proof-of-concept fires on the downstream distributions that actually ship the code [103], [104], [105], [106], [107], [114]. We include these works as triage signals (they answer "does this bug matter?") and deliberately exclude the orthogonal concern of building deployable exploits or runtime defenses, which a companion SoK covers [6].

**Root-cause analysis, the throttle.** Bridging the gap between a crash symptom and its root cause remains the hardest triage problem (C3). Prior tools operationalize root cause in three distinct categories: (i) triggering conditions that activate the bug (e.g., AURORA [83]); (ii) faulty instructions that pinpoint the defective code statement (e.g., ARCUS [84]); and (iii) vulnerability-introducing commits that identify the historical change for regression tracking (e.g., VCCFINDER [154], V-SZZ [155], and the lifetime study of Alexopoulos et al. [156]). Today, tools in all three categories remain heavyweight and offline, while direct LLM reasoning over ungrounded traces risks hallucinated explanations. Consequently, downstream patch generation (S3) cannot proceed without an actionable cause.

**Fix localization and patch–bug correlation.** A related triage task links bugs to patches. Locating the security-relevant commit for a disclosed vulnerability is itself hard, and a line of work ranks candidate fixing commits, untangles security-relevant hunks from entangled commits, and recognizes security patches in source or binaries [111], [112], [113], [115]. These tasks recur in S4 (was this bug actually fixed?) and S5 (is this patch security-relevant?), making triage and the later stages mutually dependent.

**Takeaway 2**

Triage is partially automated and partially stuck. Deduplication, impact re-ranking, severity, and exploitability estimation all have automated solutions, but root-cause analysis, the prerequisite for any repair, remains heavyweight and expert-driven at kernel scale. The gradient bends here: the pipeline can rank and label its backlog automatically, but cannot yet explain it automatically.

**Open Problem 2**

Scalable, continuous root-cause analysis is missing. Existing tools are precise but per-bug and offline; the syzbot setting needs root-cause explanations at fuzzing throughput, attached to reports automatically. Whether LLMs combined with execution traces can close this gap is open.

## 7. S3: Patch Generation

Patch generation synthesizes a candidate fix for a triaged bug. This is the stage where the automation gradient is steepest: despite a decade of automated program repair (APR) in user space, kernel-native repair is nascent, and the few systems that exist report low yields of accepted patches. We explain why the kernel is hard for repair (Section 7.1), then survey the three lines that exist (Section 7.2–7.5), deferring the rich user-space APR taxonomy to the AVR SoK [5] as contrast; representative systems for this stage are tabulated in the appendix (Section A).

### 7.1. Why the Kernel Resists Automated Repair

User-space APR assumes a property the kernel violates: a comprehensive test suite that encodes correctness, against which candidate patches can be validated cheaply [5]. The kernel offers, at best, a single crashing reproducer, and "correct behavior" is defined by implicit invariants (locking discipline, reference-count balance, memory ownership, RCU rules, challenge C2) spread across millions of lines and rarely written down. A patch must preserve these invariants under concurrency and across architectures, and a wrong patch can deadlock or silently corrupt state rather than fail a test. The test-driven generate-and-validate loop that powers user-space APR is therefore largely inapplicable, and kernel

repair has waited for techniques that can reason from context rather than from tests, which is where LLMs enter.

## 7.2. LLM Repair Agents

The current frontier is agentic LLM repair. KGYM/KBENCH provided the enabling platform that compiles, boots, and tests kernels at scale, with a dataset of real syzbot bugs and developer fixes [40]. Built on it, CRASHFIXER is the first LLM repair agent targeting the Linux kernel, mirroring a developer's investigation workflow at the scale of 20M LOC [41]; PATCHISLAND orchestrates multiple agents in a continuous-repair pipeline coupled to fuzzing [42]; and "beyond crash-to-patch" work studies how an initial fix is refined rather than one-shot generation [43]. Reported accepted-fix rates remain low (single digits on kBench-style benchmarks [40]), and most evaluations measure reproducer resolution, not upstream acceptance. In user space, by contrast, prompt-based agents already fix substantial bug counts cheaply [44], [45]. The kernel gap is one of validation infrastructure and invariants, not of generation capability per se.

## 7.3. How Far Do Current Methods Get?

To measure the generation gap directly, we benchmark thirteen LLM patch-generation configurations, eleven without and two with a localization oracle, on 80 crashes sampled from our dataset (Section 10). We restrict to evolution-stage bugs, whose first upstream fix was itself revised, so each is hard and carries review signal, and we score every candidate patch on two axes, localization and repair, the latter by a semantic judge that decides whether the patch resolves the same root cause as the merged fix. While recent work cautions that LLM-as-a-judge evaluations can introduce label inaccuracy and bias [157], we mitigate this risk through multi-model cross-validation and systematic human inspection. Two judge models score each candidate independently, each with a written rationale, and two authors re-judge every case on which the models disagree. Across a sampled subset of 100 candidate patches evaluated independently by both authors to verify agreement, author verdicts agree with the judge's on 89% of cases and with each other on 94%. The methods span one-shot prompting, sampling (Best-of-N [158]), conversational and self-directed repair (ChatRepair [44], ThinkRepair [49]), autonomous agents (RepairAgent [47], AutoCodeRover [48], RGym [46]), the CrashFixer pipeline [41], and kGym's file/function localization oracles [40], all run on the kGymSuite platform [159]. We also test two code agents at high reasoning effort, Claude Code CLI with claude-fable-5-1 and OpenAI Codex CLI with gpt-5.6-Terra.

Table 2 reports the results. We score localization at three levels across the 80 bugs: whether a patch touches at least one file from the developer fix (any), matches the exact file set (file), or matches the exact function set (func). For the last, we resolve each hunk's enclosing top-level declaration in the kernel source and score functions, struct or union definitions, and macros separately, the latter two over the 6 and 8 fixes that edit one. No configuration recovers more than 3 of the 6 struct sets or 3 of the 8 macro sets. For repair, we record the number of patches the judge rates as FIXED or PARTIAL. Threats to validity are discussed in the appendix. Localization is far easier than repair, but not solved. Methods edit at least one correct file 55-66% of the time, and a file or function level oracle pushes this to 95-100%. Pooling the eleven non-oracle configurations over all 80 bugs, only 43.5% of candidate patches recover the exact file set and 15.1% the exact function set, and on the 18 multi-file and 31 multi-function fixes no method recovers the complete set.

TABLE 2. END-TO-END LLM PATCH GENERATION ON 80 EVOLUTION-STAGE SYZBOT CRASHES.

| | Local. % | | | Local. set | | Repair (of 80) | |
|---|---|---|---|---|---|---|---|
| **Method** | **any** | **file** | **func** | **struct** | **macro** | **Fixed** | **Part.** |
| one-shot baseline | 65 | 48 | 12 | 0/6 | 0/8 | 2 | 21 |
| Best-of-N [158] | 55 | 44 | 18 | 0/6 | 1/8 | 3 | 20 |
| ChatRepair [44] | 55 | 39 | 14 | 0/6 | 0/8 | 0 | 6 |
| ThinkRepair [49] | 66 | 45 | 12 | 1/6 | 0/8 | 0 | 23 |
| RepairAgent [47] | 59 | 41 | 15 | 0/6 | 1/8 | 1 | 4 |
| AutoCodeRover [48] | 55 | 41 | 14 | 1/6 | 1/8 | 2 | 9 |
| RGym SimpleAgent [46] | 60 | 42 | 11 | 0/6 | 0/8 | 2 | 14 |
| RGym ExplorationAgent [46] | 56 | 44 | 16 | 2/6 | 2/8 | 4 | 16 |
| CrashFixer [41] | 61 | 45 | 15 | 2/6 | 3/8 | 5 | 20 |
| *recent code agents, high reasoning effort:* | | | | | | | |
| Claude Code CLI, claude-fable-5-1 | 62 | 45 | 19 | 2/6 | 3/8 | 3 | 19 |
| Codex CLI, gpt-5.6-Terra | 64 | 46 | 21 | 3/6 | 2/8 | 4 | 17 |
| *given a localization oracle (target files / functions):* | | | | | | | |
| kGym oracle, files [40] | 100 | 95 | 15 | 2/6 | 2/8 | 0 | 10 |
| kGym oracle, +functions [40] | 95 | 80 | 48 | 1/6 | 1/8 | 2 | 4 |

End-to-end repair never exceeds 5/80 (6%), and the oracle rows make the point sharpest. Told exactly which file to change, models still fix 0/80, and told the exact function, only 2/80. The wall is synthesizing a correct fix, not finding where it goes. Models mostly follow the list. Given the files, 76 of 80 candidates edit exactly those files. Given the functions, 47 do, 21 add or drop some, 11 edit none of them, and 1 returns no patch. A weaker GPT-4o-mini base repaired essentially nothing (0–1/80); only with a stronger base and real code search do the better designs begin to register. One honest caveat is that real compile-and-reproduce feedback, the engine of several of these methods, was out of reach at this scale, so the feedback-driven rows may underestimate those methods, though the 6% repair rate still shows that generating correct kernel logic remains the central bottleneck.

**Finding 1**

On 80 hard kernel crashes, LLM patch generation hits at least one correct file 55-66% of the time, and 95-100% given an oracle, but repairs at most 6%. Even told the exact file and function to edit, models fix $\leq$2/80. Kernel repair is bottlenecked on synthesizing a correct fix under implicit invariants, not on locating it.

### 7.4. Semantic Transformation

Predating LLMs, the kernel community automated repair through semantic patches. COCCINELLE's SmPL lets a maintainer express a cross-tree change as a near-patch and apply it everywhere; over a decade it is credited with thousands of commits, making it the most successful deployed kernel repair technology by volume, with roots in automating collateral evolutions as driver APIs change [27], [116]. These approaches are fully automated once a human writes the rule: they fix known patterns at scale rather than synthesizing novel fixes.

### 7.5. Backporting, the Mature Kernel Repair Task

The one kernel repair task with robust automation is backporting, which carries a mainline fix into older stable trees where names, locations, and surrounding logic differ (challenge C6). FIXMORPH synthesizes a transformation rule from a mainline patch and applies it to the older version, correctly backporting 75% of 350 patches [117]; companions classify which commits are stable-worthy, resolve conflicts against divergent downstream code, and quantify how much porting still falls to humans [118], [119], [120]. Backporting is tractable precisely because it has an oracle the rest of S3 lacks: the original patch already encodes the correct fix, so the task is transfer rather than synthesis.

**Takeaway 3**

Kernel-native patch generation is the least mature stage. The test-driven loop behind user-space APR does not transfer, because kernel correctness lives in implicit invariants rather than test suites. The only mature repair tasks are those with a built-in oracle, pattern fixing (COCCINELLE) and backporting (FIXMORPH), where a human or an existing patch supplies the specification. LLM agents are the frontier for novel fixes but report low accepted-patch yields.

**Open Problem 3**

Repair without a test oracle. The central open problem is generating kernel patches that provably preserve invariants absent comprehensive tests. This needs (i) machine-checkable encodings of kernel invariants (locking, refcount, RCU, ownership) usable as repair constraints, and (ii) benchmarks that score upstream-accepted fixes, not just reproducer resolution. Today's agents optimize the latter while ignoring the former.

## 8. S4: Patch Validation

A candidate patch, whether written by a developer or generated by an agent, is not a fix until it is shown to resolve the bug, preserve functionality, and leave no residual or newly introduced defect. Kernel validation inherits the oracle problem that hampers generation (S3): without comprehensive tests, "correct" is hard to establish mechanically, and "complete" is harder still.

**Correctness checking.** The most developed validation task asks whether a patch is correct. KLAUS attacks this directly for the kernel: from a study of 182 incorrectly developed patches, it observes that errors usually stem from the patch's altered read/write operations, and steers a fuzzer toward the affected contexts, confirming and fixing 25 incorrect patches upstream [50]. In user space, correctness assessment has a longer history [51], and LLM-as-judge schemes with a human in the loop have recently been proposed to scale it [52]. These reduce, but do not eliminate, the manual burden.

**Completeness and incomplete fixes.** Correctness is necessary but not sufficient: a patch can resolve the reported crash yet leave sibling instances unfixed, or introduce a new defect. Incomplete fixes are common enough in the kernel to be a named, studied phenomenon. Liu et al. [53] identify three recurring root causes (developers misled by the surface symptom, neglecting similar modules, or introducing a new semantic error) and build a similarity-based detector that uncovered previously unknown cases. This is precisely the failure mode automated generation (S3) is most prone to, and it is barely tooled: detecting that a fix is complete has no scalable, deployed solution.

**Patch presence testing.** Validation also has a downstream-deployment dimension: given the fragmented ecosystem of vendor and distribution kernels (challenge C6), is a particular tree actually patched? PDIFF decides whether a known fix is present despite version drift [121], and PS3 sharpens this to a precise test from a semantic signature of the patch [122]. Verifying deployment faces the same version-alignment problem as backporting (S3).

**Benchmarks for validation.** Recent work argues that validation itself needs better ground truth, formalizing patch validation around a proof-of-concept plus functional and synthesized unit tests [54]. As with generation, the scarcity of kernel benchmarks with reproducers, fixes, and completeness oracles is a limiting factor we return to in Section 10.

**Takeaway 4**

Validation is where the oracle problem bites hardest. Correctness has partial, kernel-specific automation (KLAUS) and patch-presence testing is solved (PDIFF), but completeness, did the patch fix all instances and introduce none, is essentially manual, even though it is the dominant failure mode of automatically generated patches.

**Open Problem 4**

Automated completeness checking. We lack scalable methods to decide whether a kernel patch fixes every sibling manifestation of a defect and introduces no regression. As LLM agents (S3) generate more patches, the validation bottleneck, not the generation bottleneck, will dominate. Co-designing generation with completeness-aware validation (e.g., generating the sibling-instance test alongside the patch) is unexplored.

## 9. S5: Community Integration

A validated patch is still not a deployed fix: it must be posted, reviewed, revised, accepted by a maintainer, merged, and backported to the stable trees real systems run, and even then deployment may demand a reboot that live kernel updating tries to avoid [125]. This is the automation gradient's floor: integration is governed not by an algorithm but by a socio-technical process of mailing-list review, maintainer attention, and human judgment, the stage the security literature has most neglected even as it has become the binding constraint.

**Code review.** Review is the gate every kernel patch passes through, and it is overwhelmingly manual. One line of work automates parts of it, learning the contributor and reviewer sides, pre-training on code-change/review data, and generating review comments, with the LLM era adding benchmarks, developer-aligned feedback, and workflow studies [126], [127], [128], [129], [130], [131]. Crucially, almost all of this work is evaluated on general open-source corpora, not the kernel, whose review norms (LKML etiquette, Signed-off-by chains, subsystem trees) differ sharply. A second line studies review as human practice, characterizing its expectations and outcomes, its effect on quality, reviewer participation, and review strategies [132], [133], [134], [135]; for the kernel specifically, work documents patch-submission communication [137] and shows that incivility on LKML correlates with rejected changes [136]. Integration outcomes hinge on human and social factors that no current automation models.

**The maintainer bottleneck.** The kernel's integration capacity is fundamentally a function of maintainer bandwidth, and it does not scale with the inflow of patches and bug reports. Zhou et al. [2] show that maintainer workload is highly unbalanced and that adding co-maintainers yields only sublinear gains, and Tan et al. [3] analyze the multiple-committer model's pressure–latency–quality trade-offs. The strain is visible at the margins, in newcomer onboarding, reviewer scarcity for Rust-for-Linux, and contributor retention [138], [139], [140]. After the kernel became a CVE numbering authority in 2024, CVE volume rose by an order of magnitude, sharply increasing patching demand [123]. This is the human face of the crash-to-patch gap.

**Acceptance and disclosure.** Whether and how fast a patch is accepted has been studied empirically. Jiang et al. [141] find that only a fraction of submitted patches reach a release and that author experience strongly predicts acceptance speed; coordination across CVE numbering authorities and disclosure management add further process latency [124], [142]; drivers dominate both regression frequency and fix slowness [96]; and static-analysis alerts are often left unaddressed [143]. These findings quantify, from the process side, the same delay our measurement (Section 10) observes from the data side.

**Datasets that measure the pipeline.** Finally, integration is where end-to-end datasets live. A multi-level patchwork dataset links patches, reviewers, and commits across nine years of LKML [144], and for the repair-centric pipeline, KGYM/KBENCH [40] and live crash-resolution benchmarks [55] pair reproducers with developer fixes. We use these, together with the public syzbot dashboard, as the basis for our measurement.

**Takeaway 5**

Integration is the automation-gradient floor. The decisive resources are human: maintainer attention, review latency, author reputation, even discourse civility. Automation here is nascent and, tellingly, almost never kernel-specific; the richest body of evidence is descriptive (empirical SE studies), not prescriptive.

**Open Problem 5**

Closing the loop, not just generating patches. The field optimizes generation while the binding constraint is integration: kernel-aware review assistance, maintainer-load-aware routing, and agents that carry a fix through revision rounds are all missing. Until automation targets integration, more generated patches may worsen, not relieve, the maintainer bottleneck.

## 10. Measuring the Crash-to-Patch Gap

The preceding sections argue qualitatively that automation thins toward the back of the pipeline. We now ground that claim by measuring the crash-to-patch gap on bugs that traverse the entire lifecycle, decomposing it to locate where the time is actually spent.

### 10.1. Dataset and Method

We assembled a dataset of 6,946 Linux kernel bugs that syzbot reports as fixed, each linked through its full lifecycle: first and last crash timestamps, fix timestamp and merged commit, the reconstructed patch series (v1→v2→...) and reviewer threads from `lore.kernel.org`, and reproducer availability.[3] When reconstructing review threads we discard stable-backport batch series and pull-request digests (`[PATCH 4.14 000/164]`, `[GIT PULL]`), which the archive over-associates with a bug and which would otherwise inflate per-bug discussion counts by orders of magnitude. Bug-class and subsystem labels were derived by two authors from report titles and merged-patch paths using a fixed rule set, with disagreements resolved by discussion.

We use syzbot because it provides unusually complete public linkage among crash reports, available reproducers, and fixes. Our results characterize eventually fixed, syzbot-reported bugs and may not generalize to out-of-band reports, especially those submitted with patches. We study the fixed population deliberately: these bugs have a well-defined crash-to-patch latency, and slow recent bugs are right-censored, so our latencies are a conservative lower bound on the gap. The snapshot also contains 364 still-open reports, whose open rate is highest for the bug classes

3. Collected from the public syzbot dashboard and kernel git/mail archives; scraper and analysis scripts are released with the artifact.

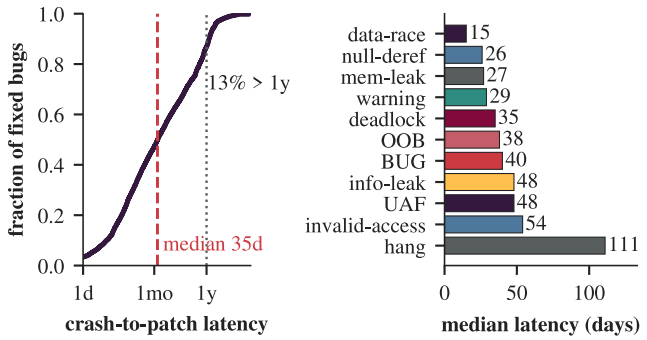


Figure 6. **Crash-to-patch latency over 6,946 fixed kernel bugs. Left: CDF on log-time. The median is 35 days but the tail is long (13% exceed one year). Right: median latency by bug class. Semantically diffuse bugs (hangs, UAF) linger, while sharp-signature bugs (races, null-deref) close fast.**

hardest to reason about (Section 10.5), reinforcing the same bias.

## 10.2. Crash-to-Patch Latency

The central measurement is the time from a bug's first observed crash to its fix. The distribution (Figure 6, left) is severe and heavy-tailed: the median fixed bug takes 35 days to patch, the mean 138; more than half take over a month, 13% take more than a year, and the slowest waited 7.5 years. For a stage that produces bugs in seconds of fuzzing, a median month-plus to patch is the automation gradient made concrete.

The latency also varies by bug class in a telling way (Figure 6, right). Semantically diffuse failures take longest, hangs/stalls (median 111 days) and corrupted-state failures (57), whose symptom sits far from its cause; classes with a sharp, local signature close fastest, data races (15) and null-pointer dereferences (26). The bugs hardest to understand (S2) and repair correctly (S3) are exactly the ones that linger, consistent with our claim that the back-end stages, not discovery, set the pace.

**Finding 2**

Even among bugs that are eventually fixed, the crash-to-patch gap is large and heavy-tailed (median 35 days, mean 138, 13% over a year), and it is a conservative lower bound: unfixed and slow recent bugs are excluded.

**The gap is structural, not transient.** One might expect a decade of improving tooling to have shrunk the gap. The temporal trend says otherwise. The very high medians of 2017–2018 (471 and 291 days) reflect syzbot's launch clearing a backlog of long-latent bugs; once the pipeline reached steady state in 2019, the median plateaued at roughly three weeks (15–25 days) and has stayed there for six consecutive years, even as discovery throughput and LLM tooling advanced. (The dip in the most recent years is right-censoring, which makes the plateau, if anything, optimistic.) Better finding has not translated into faster closing.

**Finding 3**

Since the continuous-fuzzing pipeline matured in 2019, median crash-to-patch latency has held at roughly three weeks for six years. The gap is a stable structural property of the back end, not a transient that better bug finding will erode.

## 10.3. Where the Time Goes

A single latency number cannot say which stage is slow. We therefore decompose each bug's lifecycle into ordered segments (first crash → first patch posted → final patch version → merged commit → syzbot marks fixed) using mail and git timestamps, and compute each segment's share of that bug's total latency (Figure 7, $n$=3,371 bugs with a complete, monotone chain).

First, the largest share of the wait, 51% on average, elapses before the first patch is even posted: the bug sits after discovery, waiting to be triaged, root-caused, and turned into a candidate fix. The median time to the first human reply is 6 days and to the first posted patch 7, though a heavy tail languishes for months. This is the human attention/throughput bottleneck the gradient predicts, and it dwarfs the revision loop.

Second, explicit revision churn accounts for only 4% of total latency on average, because most accepted fixes are merged at their first or second version. Revision is nonetheless the failure mode of the hard cases (21% of fixes with a reconstructable series needed two or more versions, up to nine), a thin median with a long tail. The remaining 11%, time between first posted patch and merge not explained by visible revisions, is review and acceptance latency, the patch waiting on a maintainer rather than on its author.

Third, a substantial 35% of the nominal latency is post-merge: the lag between the fix landing and syzbot confirming the bug no longer reproduces (median 7 days, mean 71). This is infrastructure latency, not engineering effort. The engineering gap (crash→merge) is therefore somewhat shorter than the headline, with the remaining delay concentrated precisely in the human-bound front of the back end, in getting a correct first patch written and landed.

**Finding 4**

About 51% of measured latency occurs before the first patch, 4% during visible revision churn, 11% during review and acceptance, and 35% during syzbot's post-merge confirmation. These timestamps locate the delay but do not identify a single cause. Bug complexity, subsystem characteristics, reproducer quality, and maintainer availability may all contribute.

## 10.4. Heterogeneity Across Subsystems

The gap is also unevenly distributed across the kernel. Among the fifteen subsystems with the most fixed bugs, median latency spans more than an order of magnitude. Here

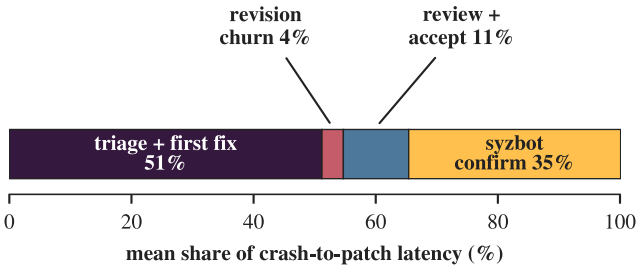


Figure 7. **Decomposition of crash-to-patch latency into ordered segments, as the mean per-bug share of total latency ($n$=3,371). Just over half the wait precedes the first posted patch (triage and first-fix authoring). Explicit revision churn is a thin 4% mean with a long tail, and a third is syzbot's post-merge confirmation lag rather than engineering time.**

`kernel/bpf` is the slowest by a wide margin (median 212 days), followed by `arch/x86` (91), `fs/ext4` (85), and cross-cutting `include/` header changes (77), while `fs/io_uring.c` (6), `net/sched` (9), `mm` (17), and `net/core` (18) close fastest. The slow subsystems are those where a fix must satisfy an unusually demanding correctness bar (the BPF verifier's safety contract) or coordinate across many drivers, whereas the fast ones tend to have a single responsive maintainer or a self-contained fix.

Reproducer availability is likewise subsystem-dependent: `net/core`, `kernel/bpf`, and `net/ipv4` bugs lack any reproducer 31-32% of the time, whereas device subsystems with concrete trigger paths (`drivers/media` 7%, `drivers/usb` 11%) are far better supplied. A repair agent's applicability is thus gated subsystem-by-subsystem by the very artifacts the front end does or does not emit.

**Finding 5**

Latency varies by more than 10× across subsystems (median 6-212 days), and reproducer scarcity ranges from 7% to 32%. Both the delay and the artifacts automation depends on are kernel-region-specific, and no single intervention closes the gap everywhere.

**Challenge prevalence.** Tagging each fixed bug shows that concurrency (C1) affects 19.3% of reports, configuration differences across trees (C6) 17.2%, and hardware dependence (C4) 16.1%, followed by implicit invariants (C2, 8.4%), cross-syscall state (C3, 7.4%), and lack of fault isolation (C5, 2.9%).

**Where the pre-merge delay sits.** Two direct signals place the engineering delay in the human-in-the-loop stages, where patches are revised and discussed over multiple rounds. Of the 5,252 bugs with a reconstructable patch series, 21% required two or more revisions before acceptance, the correctness and completeness gap of S3–S4 manifesting as resubmission, and discussion is heavy-tailed (median 6 non-bot messages, mean 18, top decile 40). Both signals sit in validation and integration, not in generating a first candidate. The remaining 35% of nominal latency is syzbot's post-merge confirmation lag (Section 10.3). Thus, our observation covers only the crash-to-merge window, where bug complexity, subsystem characteristics, and reproducer quality collectively shape the delay (Section 10.4). We therefore read the data as locating the delay, not as isolating a single cause.

**Finding 6**

The pre-merge crash-to-patch delay concentrates in the human-in-the-loop downstream stages. Some 21% of fixes take ≥2 review-driven revisions and a long tail of bugs draws 40+ reviewer messages, while the median bug spends most of its life simply waiting for a first fix. A further 35% of nominal latency is pipeline mechanism, syzbot's post-merge confirmation. This is the pattern the automation gradient predicts.

## 10.5. Automation Readiness: Reproducers and Patch Shape

Downstream automation (triage, repair agents, validation) depends on a reproducer to ground its reasoning and check its output, and is easiest when the required fix is small and local. Yet 26.7% of even the fixed bugs had no reproducer at all, and a full 33% lacked a C reproducer: a quarter of the very bugs humans did close would have been out of reach for today's reproducer-driven repair agents (S3) before any modeling limitation applies. Scarcity also tracks bug class: data races essentially never ship a reproducer (sequential replay cannot capture the interleaving), and use-after-free and hangs lack a C reproducer 40% of the time, exactly the classes whose latency is highest.

The shape of the accepted fix is more encouraging: the median merged fix touches 1 file (57% single-file) and changes 5 lines (67% change ≤10), the regime where automated repair is most plausible, so the binding constraint is the input artifacts more than the size of the edit. To make "repair-readiness" concrete, we score each fixed bug against the artifacts current agentic pipelines assume: a C reproducer, a single-file fix, a small (≤50-line) diff, and acceptance without revision. Only 34% of fixed bugs satisfy all four, and requiring light review (≤5 messages) drops the share to 18%; the rest fall outside the operating envelope today's benchmarks reward.

**Finding 7**

A quarter (26.7%) of fixed bugs lack any reproducer and a third lack a C reproducer, yet the typical fix is small (median 1 file, 5 lines). Only 34% of fixed bugs are "repair-ready" (C reproducer + single-file + ≤50-line + no revision), 18% once light review is also required. The input artifacts, not the edit size, gate back-end automation.

We also map the surveyed techniques onto a grid of bug class by lifecycle stage to see where dedicated automation exists. The matrix and its discussion are in the appendix.

**Finding 8**

No kernel bug class is served by dedicated automation across all four technical stages. Coverage is dense at discovery and triage and collapses at generation and validation. The matrix's empty lower-right region is the research frontier this SoK identifies.

## 11. Discussion and Future Directions

Our survey and measurement converge on one structural fact: automation is concentrated at discovery and drains away toward integration, and the cost of that asymmetry, the crash-to-patch gap, is dominated by the stages the security community has invested in least. We close with cross-cutting directions.

**D1. Rebalance the field from finding to closing.** The marginal discovered kernel bug is nearly free; the marginal closed bug is expensive and slow, yet discovery remains the largest category even in our balanced corpus. Effort should move toward the right half of Figure 1, and new discovery work should be evaluated on its effect on the downstream pipeline.

**D2. Downstream-aware discovery.** Discovery should surface bugs with the artifacts needed to close them: a quarter of fixed bugs lacked any reproducer, structurally blocking automated repair. Fuzzers that co-produce a minimized reproducer, root cause, or fixability estimate would attack the gap at its source.

**D3. Continuous, grounded root-cause analysis.** Scaling root-cause analysis to fuzzing throughput is the primary open challenge in triage (Section 6). To prevent ungrounded LLM reasoning and hallucinations [157], models should not guess causes in isolation. Instead, they should act as hypothesis generators coupled with dynamic execution feedback (e.g., using lightweight emulation to falsify candidate predicates). Crucially, downstream repair needs these root causes formulated as machine-actionable invariants (e.g., locking constraints or lifetime bounds) rather than human-readable text, supplying the synthesis specifications currently missing in S3 and S4.

**D4. Repair and validation without a test oracle, co-designed.** The steepest part of the gradient (S3–S4) shares one root cause: kernel correctness lives in implicit invariants, not test suites. Progress requires machine-checkable encodings of kernel invariants (locking, refcount, RCU, ownership) usable simultaneously as repair constraints and validation oracles, and generation that emits sibling-instance and regression tests alongside the patch; an agent should iterate against such oracles the way a developer iterates against reviewers.

**D5. Automation aimed at integration.** The gradient's floor (S5) is where automation is scarcest and least kernel-specific; kernel-aware review assistance, maintainer-load-aware routing, and agents that shepherd a patch through revision rounds are all open. We caution that scaling generation without scaling integration may worsen the bottleneck: machine-generated patches land on the same finite maintainer attention our data shows is already strained.

**D6. Benchmarks that score the whole lifecycle.** Current kernel benchmarks [40], [55] measure reproducer resolution, but almost none score upstream acceptance, completeness, or invariant preservation, the properties that actually gate a fix. Benchmarks that reward closing a bug as the community defines it would realign the field's incentives.

**LLMs as connective tissue.** Across stages, the four LLM roles show sharply different maturity: artifact generators are effective where they scale human expertise over a grounded artifact (specifications and checkers at S1, patches and review comments at S3/S5); classifiers/judges are a useful but unverified labeling aid (S2, S4); agents are an active but early frontier with low accepted-fix yields (S3/S5); and reasoning engines for the tasks that lack an oracle (root cause, invariant-preserving repair, completeness) remain the least mature, because the model cannot self-verify what the kernel never makes explicit. The productive frontier is the first three roles coupled to oracles, not the fourth in isolation.

**A moving snapshot.** Recent industry evidence confirms this gradient. Anthropic's Project Glasswing reported over 10,000 vulnerabilities of high or critical severity within a month, but maintainers have patched only 75 of the 530 bugs reported to them [160]. The real challenge has shifted from finding bugs to fixing them. Our automation levels capture a moving boundary rather than a permanent limit.

**Relation to prior systematizations.** Kernel-fuzzing surveys [4] organize discovery but not what follows it; the AVR SoK [5] systematizes user-space repair, whose central test-suite assumption fails in the kernel (S3); the kernel-hardening SoK [6] asks how to survive unfixed bugs where we ask how bugs get fixed; and empirical SE studies [141], [132], [133], [161], [162], [156], [163] corroborate our measurement from the process side. While Alexopoulos et al. [156] measure overall vulnerability lifetimes across open-source software, we specifically investigate post-discovery latency and the behavior of automation prerequisites across kernel lifecycle stages. To our knowledge, this is the first systematization to span the five stages together and to quantify the crash-to-patch gap as their unifying consequence.

## 12. Conclusion

We systematized the OS kernel bug lifecycle as a five-stage pipeline governed by an automation gradient: techniques are mature where bugs are found and grow sparse and human-in-the-loop toward a deployed fix, with LLMs closing the gap only where coupled to a grounded oracle. Measuring 6,946 fixed kernel bugs made the consequence concrete: a median 35-day wait that sits in the human-in-the-loop downstream stages. The community has spent a decade learning to find bugs faster than ever; the next decade's challenge, and this SoK's call, is to learn to close them.

## Acknowledgment

We thank the anonymous reviewers and our shepherd for their comments. This work is supported in part by US NSF grants 2127491, 2154199, and a Google Cloud gift award. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

## References

[1] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," BlueHat IL, https://github.com/microsoft/MSRC-Security-Research, 2019.

[2] M. Zhou, Q. Chen, A. Mockus, and F. Wu, "On the scalability of Linux kernel maintainers' work," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 27–37.

[3] X. Tan, M. Zhou, and B. Fitzgerald, "Scaling open source communities: An empirical study of the Linux kernel," in *IEEE/ACM International Conference on Software Engineering*, 2020, pp. 1222–1234.

[4] J. Xu *et al.*, "A survey of operating system kernel fuzzing," *ACM Transactions on Software Engineering and Methodology*, 2025, just Accepted.

[5] Y. Li, F. H. Shezan, B. Wei, G. Wang, and Y. Tian, "SoK: Towards effective automated vulnerability repair," in *USENIX Security Symposium*, 2025, pp. 4441–4462.

[6] Y. Hu, P. Ding, Z. Lin, D. Mu, and Y. Li, "SoK: Take a deep step into Linux kernel hardening effectiveness from the offensive-defensive perspective," in *Network and Distributed System Security Symposium*, 2026.

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

[8] D. R. Jeong, B. Lee, I. Shin, and Y. Kwon, "SegFuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2023, pp. 2104–2121.

[9] S. Gong, D. Altınbüken, P. Fonseca, and P. Maniatis, "Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis," in *ACM SIGOPS Symposium on Operating Systems Principles*, 2021, pp. 66–83.

[10] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency Use-After-Free bugs in Linux device drivers," in *USENIX Annual Technical Conference*, 2019, pp. 255–268.

[11] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the Linux kernel," in *USENIX Security Symposium*, 2017, pp. 1–16.

[12] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "KRACE: Data race fuzzing for kernel file systems," in *IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 1643–1660.

[13] S. Gong, D. Peng, D. Altinbüken, P. Fonseca, and P. Maniatis, "Snowcat: Efficient kernel concurrency testing using a learned coverage predictor," in *ACM SIGOPS Symposium on Operating Systems Principles*, 2023, pp. 35–51.

[14] L. Ma *et al.*, "When top-down meets bottom-up: Detecting and exploiting Use-After-Cleanup bugs in Linux kernel," in *IEEE Symposium on Security and Privacy (S&P)*, 2023, pp. 2138–2154.

[15] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in OS kernels," in *IEEE Symposium on Security and Privacy (S&P)*, 2018, pp. 661–678.

[16] S. Kim *et al.*, "Finding semantic bugs in file systems with an extensible fuzzing framework," in *ACM SIGOPS Symposium on Operating Systems Principles*, 2019, pp. 147–161.

[17] S. Bai, Z. Zhang, and H. Hu, "CountDown: Refcount-guided fuzzing for exposing temporal memory errors in Linux kernel," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24, 2024, pp. 1315–1329.

[18] A. Machiry *et al.*, "DR. CHECKER: A soundy analysis for Linux kernel drivers," in *USENIX Security Symposium*, 2017, pp. 1007–1024.

[19] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-Miner: Uncovering memory corruption in Linux," in *Network and Distributed System Security Symposium*, 2018.

[20] K. Lu, A. Pakki, and Q. Wu, "Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences," in *USENIX Security Symposium*, 2019, pp. 1769–1786.

[21] W. Wang, K. Lu, and P.-C. Yew, "Check it again: Detecting lacking-recheck bugs in OS kernels," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018, pp. 1899–1913.

[22] Y. Zhai *et al.*, "UBITect: A precise and scalable method to detect use-before-initialization bugs in Linux kernel," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 221–232.

[23] N. Emamdoost, Q. Wu, K. Lu, and S. McCamant, "Detecting kernel memory leaks in specialized modules with ownership reasoning," in *Network and Distributed System Security Symposium*, 2021.

[24] Y. Lyu *et al.*, "Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis," in *IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 2096–2113.

[25] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: Analyzing the impact of undefined behavior," in *ACM SIGOPS Symposium on Operating Systems Principles*, 2013, pp. 260–275.

[26] C. Yang, Z. Zhao, Z. Xie, H. Li, and L. Zhang, "KNighter: Transforming static analysis with LLM-synthesized checkers," in *ACM SIGOPS Symposium on Operating Systems Principles*, 2025, pp. 655–669.

[27] J. Lawall and G. Muller, "Coccinelle: 10 years of automated evolution in the Linux kernel," in *USENIX Annual Technical Conference*, 2018, pp. 601–614.

[28] T. Li, J.-J. Bai, Y. Sui, and S.-M. Hu, "Path-sensitive and alias-aware typestate analysis for detecting OS bugs," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 859–872.

[29] C. Liu, Y. Chen, and L. Lu, "KUBO: Precise and scalable detection of user-triggerable undefined behavior bugs in OS kernel," in *Network and Distributed System Security Symposium*, 2021.

[30] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An LLM-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.

[31] D. Liu *et al.*, "Detecting kernel memory bugs through inconsistent memory management intention inferences," in *USENIX Security Symposium*, 2024, pp. 4069–4086.

[32] Y. Yang *et al.*, "Making memory account accountable: Analyzing and detecting memory missing-account bugs for container platforms," in *Annual Computer Security Applications Conference (ACSAC)*, 2022, pp. 869–880.

[33] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: A Pointer-Analysis-based static approach for detecting Use-After-Free vulnerabilities," in *IEEE/ACM International Conference on Software Engineering*, 2018, pp. 327–337.

[34] X. Tan, Y. Zhang, X. Yang, K. Lu, and M. Yang, "Detecting kernel refcount bugs with two-dimensional consistency checking," in *USENIX Security Symposium*, 2021, pp. 2471–2488.

[35] T. Zhang *et al.*, "PeX: A permission check analysis framework for Linux kernel," in *USENIX Security Symposium*, 2019, pp. 1205–1220.

[36] K. Lu, A. Pakki, and Q. Wu, "Automatically identifying security checks for detecting kernel semantic bugs," in *European Symposium on Research in Computer Security (ESORICS)*, 2019, pp. 3–25.

[37] N. Dossche and B. Coppens, "Inference of error specifications and bug detection using structural similarities," in *USENIX Security Symposium*, 2024, pp. 1885–1902.

[38] H. Zhong, X. Wang, and H. Mei, "Inferring bug signatures to detect real bugs," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 571–584, 2022.

[39] B. Garmany, M. Stoffel, R. Gawlik, and T. Holz, "Static detection of uninitialized stack variables in binary code," in *European Symposium on Research in Computer Security (ESORICS)*, 2019, pp. 68–87.

[40] A. Mathai, C. Huang, P. Maniatis, A. Nogikh, F. Ivančić, J. Yang, and B. Ray, "kGym: A platform and dataset to benchmark large language models on Linux kernel crash resolution," in *Conference on Neural Information Processing Systems, Datasets and Benchmarks Track (NeurIPS)*, 2024.

[41] A. Mathai, C. Huang, S. Ma, J. Kim, H. Mitchell, A. Nogikh, P. Maniatis, F. Ivančić, J. Yang, and B. Ray, "CrashFixer: A crash resolution agent for the Linux kernel," 2025.

[42] W. Kim *et al.*, "PatchIsland: Orchestration of LLM agents for continuous vulnerability repair," 2026.

[43] L. Bai, K. Alghythee, H. Zhang, and X. Wang, "Beyond crash-to-patch: Patch evolution for Linux kernel repair," 2026.

[44] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 819–831.

[45] M. Böhme, C. Geethal, and V.-T. Pham, "Human-in-the-loop automatic program repair," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2020, pp. 274–285.

[46] K. Shehada *et al.*, "Rethinking kernel program repair: Benchmarking and enhancing LLMs with RGym," 2025, conference on Neural Information Processing Systems (NeurIPS) Workshop: Evaluating the Evolving LLM Lifecycle.

[47] I. Bouzenia, P. T. Devanbu, and M. Pradel, "RepairAgent: An autonomous, LLM-based agent for program repair," in *IEEE/ACM International Conference on Software Engineering*, 2025, pp. 2188–2200.

[48] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "AutoCodeRover: Autonomous program improvement," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1592–1604.

[49] X. Yin *et al.*, "ThinkRepair: Self-directed automated program repair," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1274–1286.

[50] Y. Wu *et al.*, "Mitigating security risks in Linux with KLAUS: A method for evaluating patch correctness," in *USENIX Security Symposium*, 2023, pp. 4247–4264.

[51] A. Ghanbari and A. Marcus, "Patch correctness assessment in automated program repair based on the impact of patches on production and test code," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 654–665.

[52] S. Shi, R. Wei, M. Tufano, J. Cambronero, R. Cheng, F. Ivančić, and P. Rondon, "Towards a human-in-the-loop framework for reliable patch evaluation using an LLM-as-a-judge," 2025.

[53] Q. Liu, W. Zhang, M. Jiang, L. Wu, and Y. Zhou, "Characteristics, root causes, and detection of incomplete security bug fixes in the Linux kernel," 2025.

[54] Z. Yu *et al.*, "Patch validation in automated vulnerability repair," 2026.

[55] C. Huang, A. Mathai, F. Yu, A. Nogikh, P. Maniatis, F. Ivančić, E. Wu, K. Kaffes, J. Yang, and B. Ray, "Outrunning LLM cutoffs: A live kernel crash resolution benchmark for all," 2026.

[56] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted feedback fuzzing for OS kernels," in *USENIX Security Symposium*, 2017, pp. 167–182.

[57] J. Liu, Y. Shen, Y. Xu, H. Sun, and Y. Jiang, "Horus: Accelerating kernel fuzzing through efficient host-VM memory access procedures," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 11:1–11:25, 2023.

[58] M. Cho, D. An, H. Jin, and T. Kwon, "BoKASAN: Binary-only kernel address sanitizer for effective kernel fuzzing," in *USENIX Security Symposium*, 2023, pp. 4985–5002.

[59] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS fuzzer seed selection with trace distillation," in *USENIX Security Symposium*, 2018, pp. 729–743.

[60] H. Sun *et al.*, "HEALER: Relation learning guided kernel fuzzing," in *ACM SIGOPS Symposium on Operating Systems Principles*, 2021, pp. 344–358.

[61] J. Xu *et al.*, "MOCK: Optimizing kernel fuzzing mutation with context-aware dependency," in *Network and Distributed System Security Symposium*, 2024.

[62] M. Fleischer *et al.*, "ACTOR: Action-guided kernel fuzzing," in *USENIX Security Symposium*, 2023, pp. 5003–5020.

[63] D. Wang *et al.*, "SyzVegas: Beating kernel fuzzing odds with reinforcement learning," in *USENIX Security Symposium*, 2021, pp. 2741–2758.

[64] S. Gong, R. Wang, D. Altınbüken, P. Fonseca, and P. Maniatis, "Snowplow: Effective kernel fuzzing with a learned White-box test mutator," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '25, 2025, pp. 1124–1138.

[65] H. Han and S. K. Cha, "IMF: Inferred model-based fuzzer," in *ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2345–2358.

[66] W. Chen, Y. Wang, Z. Zhang, and Z. Qian, "SyzGen: Automated generation of syscall specification of closed-source macOS drivers," in *ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 749–763.

[67] H. Sun, Y. Shen, J. Liu, Y. Xu, and Y. Jiang, "KSG: Augmenting kernel fuzzing with system call specification generation," in *USENIX Annual Technical Conference*, 2022, pp. 351–366.

[68] Y. Hao *et al.*, "SyzDescribe: Principled, automated, static generation of syscall descriptions for kernel drivers," in *IEEE Symposium on Security and Privacy (S&P)*, 2023, pp. 3262–3278.

[69] C. Yang, Z. Zhao, and L. Zhang, "KernelGPT: Enhanced kernel fuzzing via large language models," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '25, 2025, pp. 560–573.

[70] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *IEEE Symposium on Security and Privacy (S&P)*, 2019, pp. 818–834.

[71] B. Zhao *et al.*, "StateFuzz: System Call-Based State-Aware Linux driver fuzzing," in *USENIX Security Symposium*, 2022, pp. 3273–3289.

[72] K. Kim *et al.*, "HFL: Hybrid fuzzing on the Linux kernel," in *Network and Distributed System Security Symposium*, 2020.

[73] X. Tan *et al.*, "SyzDirect: Directed greybox fuzzing for Linux kernel," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23, 2023, pp. 1630–1644.

[74] D. Vyukov and A. Konovalov, "Syzkaller: An unsupervised coverage-guided kernel fuzzer," https://github.com/google/syzkaller, 2015.

[75] J. Liu, Y. Shen, Y. Xu, and Y. Jiang, "Leveraging binary coverage for effective generation guidance in kernel fuzzing," in *ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 3763–3777.

[76] W. Chen *et al.*, "SyzGen++: Dependency inference for augmenting kernel driver fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2024, pp. 4661–4677.

[77] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "No grammar, no problem: Towards fuzzing the Linux kernel without system-call descriptions," in *Network and Distributed System Security Symposium*, 2023.

[78] H. Zhang *et al.*, "Statically discovering high-order taint style vulnerabilities in OS kernels," in *ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 811–824.

[79] H. Li, H. Zhang, K. Pei, and Z. Qian, "Towards more accurate static analysis for Taint-Style bug detection in Linux kernel," in *IEEE/ACM International Conference on Automated Software Engineering*, 2025, pp. 380–392.

[80] H. Zhang, J. Kim, C. Yuan, Z. Qian, and T. Kim, "Statically discover Cross-Entry Use-After-Free vulnerabilities in the Linux kernel," in *Network and Distributed System Security Symposium*, 2025.

[81] D. Mu *et al.*, "An in-depth analysis of duplicated Linux kernel bug reports," in *Network and Distributed System Security Symposium*, 2022.

[82] J. Bursey, A. A. Sani, and Z. Qian, "SyzRetrospector: A large-scale retrospective study of syzbot," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2025, pp. 92–105.

[83] T. Blazytko *et al.*, "AURORA: Statistical crash analysis for automated root cause explanation," in *USENIX Security Symposium*, 2020, pp. 235–252.

[84] C. Yagemann *et al.*, "ARCUS: Symbolic root cause analysis of exploits in production systems," in *USENIX Security Symposium*, 2021, pp. 1989–2006.

[85] Z. Jiang *et al.*, "Igor: Crash deduplication through Root-Cause clustering," in *ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3318–3336.

[86] D. C. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of emulation for kernelspace fuzzing," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2019.

[87] D. Song *et al.*, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *USENIX Security Symposium*, 2020, pp. 2541–2557.

[88] J. Corina *et al.*, "DIFUZE: Interface aware fuzzing for kernel drivers," in *ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[89] W. Zhao, K. Lu, Q. Wu, and Y. Qi, "Semantic-informed driver fuzzing without both the hardware devices and the emulators," in *Network and Distributed System Security Symposium*, 2022.

[90] Z. Ma *et al.*, "PrIntFuzz: Fuzzing Linux drivers via automated virtual device simulation," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 404–416.

[91] T. Yin *et al.*, "KextFuzz: Fuzzing macOS kernel EXTensions on apple silicon via exploiting mitigations," in *USENIX Security Symposium*, 2023, pp. 5039–5054.

[92] J. Choi, K. Kim, D. Lee, and S. K. Cha, "NTFUZZ: Enabling type-aware kernel fuzzing on Windows with static binary analysis," in *IEEE Symposium on Security and Privacy (S&P)*, 2021, pp. 677–693.

[93] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *USENIX Security Symposium*, 2021, pp. 2597–2614.

[94] H. Peng and M. Payer, "USBFuzz: A framework for fuzzing USB drivers by device emulation," in *USENIX Security Symposium*, 2020, pp. 2559–2575.

[95] J. Jang, M. Kang, and D. Song, "ReUSB: Replay-guided USB driver fuzzing," in *USENIX Security Symposium*, 2023, pp. 2921–2938.

[96] J. Ruohonen and A. Alami, "Fast fixes and faulty drivers: An empirical analysis of regression bug fixing times in the Linux kernel," 2024.

[97] J. Pan, G. Yan, and X. Fan, "Digtool: A virtualization-based framework for detecting kernel vulnerabilities," in *USENIX Security Symposium*, 2017, pp. 149–165.

[98] K. Man *et al.*, "SCAD: Towards a universal and automated network side-channel vulnerability detection," in *IEEE Symposium on Security and Privacy (S&P)*, 2025, pp. 1861–1876.

[99] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, "SyzScope: Revealing high-risk security impacts of fuzzer-exposed bugs in Linux kernel," in *USENIX Security Symposium*, 2022, pp. 3201–3217.

[100] Q. Wu, Y. Xiao, X. Liao, and K. Lu, "OS-aware vulnerability prioritization via differential severity analysis," in *USENIX Security Symposium*, 2022, pp. 395–412.

[101] M. J. Torkamani *et al.*, "Streamlining security vulnerability triage with large language models," 2025.

[102] M. Jimenez, M. Papadakis, and Y. L. Traon, "Vulnerability prediction models: A case study on the Linux kernel," in *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016, pp. 1–10.

[103] Z. Lin *et al.*, "GREBE: Unveiling exploitation potential for Linux kernel bugs," in *IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 2078–2095.

[104] W. Chen, X. Zou, G. Li, and Z. Qian, "KOOBE: Towards facilitating exploit generation of kernel Out-Of-Bounds write vulnerabilities," in *USENIX Security Symposium*, 2020, pp. 1093–1110.

[105] Z. Liang, X. Zou, C. Song, and Z. Qian, "K-LEAK: Towards automating the generation of multi-step infoleak exploits against the Linux kernel," in *Network and Distributed System Security Symposium*, 2024.

[106] W. Wu *et al.*, "FUZE: Towards facilitating exploit generation for kernel Use-After-Free vulnerabilities," in *USENIX Security Symposium*, 2018, pp. 781–797.

[107] W. Wu, Y. Chen, X. Xing, and W. Zou, "KEPLER: Facilitating control-flow hijacking primitive evaluation for Linux kernel vulnerabilities," in *USENIX Security Symposium*, 2019, pp. 1187–1204.

[108] G. Lee, D. Xu, S. Salimi, B. Lee, and M. Payer, "SyzRisk: A change-pattern-based continuous kernel regression fuzzer," in *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, ser. AsiaCCS '24, 2024, pp. 1480–1494.

[109] Y. Zhai *et al.*, "Progressive scrutiny: Incremental detection of UBI bugs in the Linux kernel," in *Network and Distributed System Security Symposium*, 2022.

[110] J. Oh, N. F. Yıldıran, J. Braha, and P. Gazzillo, "Finding broken Linux configuration specifications by statically analyzing the Kconfig language," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, pp. 893–905.

[111] X. Tan *et al.*, "Locating the security patches for disclosed OSS vulnerabilities with vulnerability-commit correlation ranking," in *ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3282–3299.

[112] S. Sun *et al.*, "DisPatch: Unraveling security patches from entangled code changes," in *USENIX Security Symposium*, 2025, pp. 4521–4540.

[113] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "SPAIN: Security patch analysis for binaries towards understanding the pain and pills," in *IEEE/ACM International Conference on Software Engineering*, 2017, pp. 462–472.

[114] X. Zou *et al.*, "SyzBridge: Bridging the gap in exploitability assessment of Linux kernel bugs in the Linux ecosystem," in *Network and Distributed System Security Symposium*, 2024.

[115] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, "SPI: Automated identification of security patches via commits," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 13:1–13:27, 2022.

[116] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," in *European Conference on Computer Systems*, 2008, pp. 247–260.

[117] R. Shariffdeen *et al.*, "Automated patch backporting in Linux (experience paper)," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 633–645.

[118] T. Hoang, J. Lawall, Y. Tian, R. J. Oentaryo, and D. Lo, "PatchNet: Hierarchical deep learning-based stable patch identification for the Linux kernel," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2471–2486, 2021.

[119] R. Liu *et al.*, "Patchscope: LLM-enhanced fine-grained stable patch classification for Linux kernel," *Proceedings of the ACM on Software Engineering (PACMSE, ISSTA)*, vol. 2, no. ISSTA, pp. 1513–1535, 2025.

[120] X. Li, Z. Zhang, Z. Qian, T. Jaeger, and C. Song, "An investigation of patch porting practices of the Linux kernel ecosystem," in *IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2024, pp. 63–74.

[121] Z. Jiang *et al.*, "PDiff: Semantic-based patch presence testing for downstream kernels," in *ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1149–1163.

[122] Q. Zhan *et al.*, "PS3: Precise patch presence test based on semantic symbolic signature," in *IEEE/ACM International Conference on Software Engineering*, 2024, pp. 167:1–167:12.

[123] G. Kudrjavets, "Patch me if you can—securing the Linux kernel," in *IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2025, pp. 142–143.

[124] J. Lin, B. Adams, and A. E. Hassan, "On the coordination of vulnerability fixes: An empirical study of practices from 13 CVE numbering authorities," *Empirical Software Engineering*, vol. 28, no. 6, p. 151, 2023.

[125] M. Siniavine and A. Goel, "Seamless kernel updates," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.

[126] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *IEEE/ACM International Conference on Software Engineering*, 2021, pp. 163–174.

[127] Z. Li *et al.*, "Automating code review activities by large-scale pre-training," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022, pp. 1035–1047.

[128] L. Li *et al.*, "AUGER: Automatically generating review comments with pre-training models," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022, pp. 1009–1021.

[129] Z. Zeng *et al.*, "Benchmarking and studying the LLM-based code review," 2025.

[130] Z. Fang, Y. Zhang, Y. Zhang, K. Leach, and Y. Huang, "DPO-F+: Aligning code repair feedback with developers' preferences," 2025.

[131] F. S. Aðalsteinsson, B. B. Magnússon, M. Milicevic, A. N. Davidsson, and C.-H. Cheng, "Rethinking code review workflows with LLM assistance: An empirical study," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2025, pp. 488–497.

[132] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *IEEE/ACM International Conference on Software Engineering*, 2013, pp. 712–721.

[133] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.

[134] S. Ruangwan, P. Thongtanunam, A. Ihara, and K. Matsumoto, "The impact of human factors on the participation decision of reviewers in modern code review," *Empirical Software Engineering*, vol. 24, no. 2, pp. 973–1016, 2019.

[135] P. W. Gonçalves, E. Fregnan, T. Baum, K. Schneider, and A. Bacchelli, "Do explicit review strategies improve code review performance? towards understanding the role of cognitive load," *Empirical Software Engineering*, vol. 27, no. 4, p. 99, 2022.

[136] I. Ferreira, J. Cheng, and B. Adams, "The "shut the f**k up" phenomenon: Characterizing incivility in open source code review discussions," *Proceedings of the ACM on Human-Computer Interaction (CSCW)*, vol. 5, no. CSCW2, pp. 353:1–353:35, 2021.

[137] X. Tan and M. Zhou, "How to communicate when submitting patches: An empirical study of the Linux kernel," *Proceedings of the ACM on Human-Computer Interaction (CSCW)*, vol. 3, no. CSCW, pp. 108:1–108:26, 2019.

[138] A. K. Turzo, S. Sultana, and A. Bosu, "From first patch to long-term contributor: Evaluating onboarding recommendations for OSS newcomers," 2024.

[139] H. Li, L. Guo, Y. Yang, S. Wang, and M. Xu, "An empirical study of Rust-for-Linux: The success, dissatisfaction, and compromise," in *USENIX Annual Technical Conference*, 2024, pp. 425–443.

[140] B. Trinkenreich, "Please don't go – increasing women's participation in open source software," in *IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE Companion)*, 2021, pp. 138–140.

[141] Y. Jiang, B. Adams, and D. M. German, "Will my patch make it? and how fast? case study on the Linux kernel," in *IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2013, pp. 101–110.

[142] S. Liu *et al.*, "An empirical study on vulnerability disclosure management of open source software systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 34, no. 7, pp. 214:1–214:31, 2025.

[143] N. Imtiaz, B. Murphy, and L. Williams, "How do developers act on static analysis alerts? an empirical study of Coverity usage," in *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 323–333.

[144] Y. Xu and M. Zhou, "A multi-level dataset of Linux kernel patchwork," in *IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2018, pp. 54–57.

[145] N. Palix *et al.*, "Faults in Linux: Ten years later," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 305–318.

[146] N. Palix *et al.*, "Faults in Linux 2.6," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 4:1–4:40, 2014.

[147] Google, "syzbot: Continuous fuzzing dashboard for the Linux kernel," https://syzkaller.appspot.com, 2026.

[148] Y. Hao *et al.*, "Demystifying the dependency challenge in kernel fuzzing," in *IEEE/ACM International Conference on Software Engineering*, 2022, pp. 659–671.

[149] H. Shi *et al.*, "Industry practice of directed kernel fuzzing for open-source Linux distribution," in *IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2159–2169.

[150] H. Shi *et al.*, "Industry practice of coverage-guided enterprise Linux kernel fuzzing," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 986–995.

[151] J. Ruohonen and K. Rindell, "Empirical notes on the interaction between continuous kernel fuzzing and development," in *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019, pp. 276–281.

[152] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 590–604.

[153] Z. Li *et al.*, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Network and Distributed System Security Symposium*, 2018.

[154] H. Perl *et al.*, "VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 426–437.

[155] L. Bao, X. Xia, A. E. Hassan, and X. Yang, "V-SZZ: Automatic identification of version ranges affected by CVE vulnerabilities," in *IEEE/ACM International Conference on Software Engineering*, 2022, pp. 2352–2364.

[156] N. Alexopoulos, M. Brack, J. P. Wagner, T. Grube, and M. Mühlhäuser, "How long do vulnerabilities live in the code? a large-scale empirical measurement study on FOSS vulnerability lifetimes," in *USENIX Security Symposium*, 2022, pp. 359–376.

[157] J. Evertz *et al.*, "Chasing shadows: Pitfalls in LLM security research," in *Network and Distributed System Security Symposium*, 2026.

[158] X. Wang *et al.*, "Self-consistency improves chain of thought reasoning in language models," in *International Conference on Learning Representations (ICLR)*, 2023.

[159] ARiSE-Lab, "kGymSuite: A distributed kernel build, test, and crash reproduction framework," https://github.com/ARiSE-Lab/kGymSuite, 2024.

[160] Anthropic, "Project Glasswing: An initial update," https://www.anthropic.com/research/glasswing-initial-update, 2026.

[161] H. Chen *et al.*, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*, 2011, pp. 1–5.

[162] M. Jimenez, M. Papadakis, and Y. L. Traon, "An empirical analysis of vulnerabilities in OpenSSL and the Linux kernel," in *Asia-Pacific Software Engineering Conference (APSEC)*, 2016, pp. 105–112.

[163] L. He, P. Su, C. Zhang, Y. Cai, and J. Ma, "One simple API can cause hundreds of bugs: An analysis of refcounting bugs in all modern Linux kernels," in *ACM SIGOPS Symposium on Operating Systems Principles*, 2023, pp. 52–65.

# Appendix A. Classification Tables

**A coverage-gap matrix.** Our final empirical instrument maps the surveyed techniques onto a (bug class × lifecycle stage) grid (Table 3), marking each cell by the strongest automation available: a dedicated kernel technique (●), only a generic one (◐), or none (○). The shape mirrors the gradient exactly. Discovery is uniformly covered; triage has dedicated exploitability tooling for memory bugs but relies on generic root-cause analysis elsewhere; generation and validation are mostly ◐/○, and no bug class enjoys end-to-end dedicated automation. The empty lower-right of the matrix is the crash-to-patch gap, drawn at the technique level.

TABLE 3. COVERAGE-GAP MATRIX: STRONGEST AUTOMATION PER (BUG CLASS × TECHNICAL STAGE). ●=DEDICATED KERNEL TECHNIQUE, ◐=GENERIC ONLY, ○=NONE; COUNTS ARE FIXED-BUG FREQUENCIES IN OUR DATASET.

| Bug class | # | Disc. | Triage | Gen. | Valid. |
|---|---|---|---|---|---|
| UAF | 979 | ● | ● | ◐ | ◐ |
| OOB | 633 | ● | ● | ◐ | ◐ |
| uninit (UBI) | 497 | ● | ◐ | ◐ | ◐ |
| null-deref | 592 | ● | ◐ | ◐ | ◐ |
| GPF | 524 | ● | ◐ | ◐ | ◐ |
| data-race | 199 | ● | ◐ | ○ | ○ |
| deadlock/lock | 470 | ● | ◐ | ○ | ○ |
| hang/stall | 351 | ● | ◐ | ○ | ○ |
| mem-leak | 243 | ● | ● | ◐ | ◐ |
| WARNING/other | 2238 | ● | ◐ | ◐ | ○ |

Table 4 lists representative systems for stages S2 to S5 with their automation level, and Table 5 gives the full discovery classification.

TABLE 4. REPRESENTATIVE TECHNIQUES, STAGES S2–S5 (TABLE 5 COVERS S1). *Auto.*: A0 PRODUCTION-DEPLOYED, A1 OFFLINE/PROTOTYPE, H HUMAN-IN-LOOP, M MANUAL. ★ LLM-BASED, (U) USER-SPACE CONTRAST.

| System | Focus | Auto. | LLM |
|---|---|---|---|
| S2 *Triage* | | | |
| SyzScope [99] | impact re-rank | A1 | |
| LLM triage [101] | severity | A1 | ★ |
| GREBE [103] | exploitability | A1 | |
| AURORA [83] | root cause | A1 | |
| ARCUS [84] | root cause | A1 | |
| S3 *Generation* | | | |
| CrashFixer [41] | LLM agent | A1 | ★ |
| PatchIsland [42] | LLM agent | A1 | ★ |
| kGym/kBench [40] | platform | A1 | ★ |
| Coccinelle [27] | semantic | A0 | |
| FixMorph [117] | backport | A1 | |
| PatchNet [118] | backport sel. | A1 | |
| S4 *Validation* | | | |
| KLAUS [50] | correctness | H | |
| LLM-judge [52] | correctness (U) | H | ★ |
| Incomplete-fix [53] | completeness | A1 | |
| PDiff [121] | presence | A1 | |
| S5 *Integration* | | | |
| CodeReviewer [127] | review autom. | A1 | ★ |
| Zhou et al. [2] | maintainer load | M | |
| Rust-for-Linux [139] | reviewer scarce | M | |
| Patch-Me [123] | CVE flood | M | |
| Jiang et al. [141] | acceptance | M | |
| Patchwork [144] | dataset | M | |

**Table 5** The Tech. column lists each system's technique families, primary first. Dynamic families are coverage or execution (Cov), input-structure or specification inference (Inp), dependency or sequence inference (Dep), mutation or task scheduling (Sch), state-aware fuzzing (Sta), directed fuzzing (Dir), concurrency interleaving (Con), and device emulation (Emu). Static families are taint or dataflow anal-

ysis (Tnt), typestate or lifecycle analysis (Typ), symbolic or path-sensitive reasoning (Sym), specification or pattern mining (Min), and learned models (ML).

**Threats to validity.** Our measurement studies the fixed population from one (dominant) source, syzbot. Bugs reported elsewhere often arrive with a patch attached, skipping the interval we measure, so our figures characterize the fuzzer-found population, and right-censoring understates latency, both conservative with respect to our thesis. Two authors independently labeled all 110 systems. Agreement was 88.2% ($\kappa = 0.81$) for stage, 81.8% ($\kappa = 0.72$) for challenge, and 83.6% ($\kappa = 0.75$) for automation level. Because a system can carry several challenge labels, a system counts as agreed on challenge only when both authors assign the same set of labels, and $\kappa$ is computed on that set-level judgment. Disagreements concentrated on the A1 versus H boundary and were resolved by discussion, with the stricter label chosen when a tool proposes but a human decides. Reassigning every disputed label, whether stage, challenge, or the A1 versus H boundary, moves no technique across the front-end and back-end divide, so neither the automation gradient nor the coverage-gap matrix changes. Patch-series reconstruction from mailing archives is incomplete, so revision and review-effort figures are lower bounds, and $\sim$35% of our headline latency is post-merge confirmation lag rather than engineering time (Section 10.3). Our benchmark and LLM-assisted labeling face the pitfalls of LLM-based security evaluation catalogued by Evertz et al. [157], including training-data contamination, prompt sensitivity, reliance on an LLM judge, and absent execution feedback. We mitigate these with recorded model versions, two judge models with author re-judging of disagreements and an independent author check on 100 sampled candidates, and by releasing every verdict and rationale for audit (Section 7.3). On the buildable subset, the judge's verdicts matched real rebuild-and-reproduce outcomes in 32 of 36 cases. A different lens on the same papers is possible.

## Appendix B. Open Science

We release the full dataset of 6,946 syzbot-fixed Linux kernel bugs, the analysis scripts behind every figure and statistic, the classification of the 140 surveyed papers, and the judge rubric, verdicts, and rationales from Section 7.3.

## Appendix C. Ethics Considerations

The measurement in Section 10 and the benchmark in Section 7.3 use only public data on kernel bugs that syzbot had already marked as fixed, drawn from the public syzbot dashboard, mailing-list archives, and kernel git history. No unfixed vulnerability, proof of concept, or exploit is included or produced. The LLM-generated candidate patches were evaluated offline against the merged developer fixes and were not submitted to kernel maintainers. Developer names appear in the released records only as in the public archives, and the paper reports aggregate statistics only.

TABLE 5. FULL DISCOVERY (S1) CLASSIFICATION. *Type*: F=FUZZING, S=STATIC, H=HYBRID, E=EMPIRICAL STUDY. *Bug class*: MEMORY (MEM), CONCURRENCY (CONC), LOGIC, SEMANTIC (SEM), TAINT. ★ LLM-BASED.

| System | Type | Target | Bug class | Tech. | LLM |
|---|---|---|---|---|---|
| *Dynamic discovery (fuzzing)* | | | | | |
| kAFL [56] | F | generic | mem | Cov | |
| Unicorefuzz [86] | F | generic | mem | Cov | |
| Agamotto [87] | F | driver | mem | Cov, Emu | |
| Horus [57] | F | generic | mem | Cov | |
| BoKASAN [58] | F | generic | mem | Cov | |
| MoonShine [59] | F | core | mem | Dep, Cov, Tnt | |
| HEALER [60] | F | core | mem | Dep, Cov | |
| ACTOR [62] | F | core | mem | Sta, Dep, Tnt | |
| MOCK [61] | F | core | mem | Dep, Sch, ML, Cov | |
| SyzVegas [63] | F | core | mem | Sch, Cov | |
| Snowplow [64] | F | core | mem | Sch, ML, Dir, Cov | |
| StateFuzz [71] | F | driver | mem,logic | Sta, Sym, Sch | |
| HFL [72] | H | core | mem | Sym, Dep, Inp, Tnt, Sta | |
| CountDown [17] | F | core | mem (refcnt) | Sta, Dep, Sch | |
| Bin-Cov [75] | F | generic | mem | Cov, Tnt | |
| DIFUZE [88] | F | driver | mem | Inp, Sym, Tnt | |
| IMF [65] | F | API (macOS) | mem | Dep, Inp, Min | |
| DR. FUZZ [89] | F | driver | mem | Inp, Sta, Sch, Tnt, Cov | |
| PrIntFuzz [90] | F | driver | mem | Emu, Inp, Tnt, Sym, Cov | |
| KextFuzz [91] | F | drv (macOS) | mem | Cov, Inp, Tnt, Dep | |
| NTFUZZ [92] | F | drv (Win) | mem | Inp, Tnt, Sch | |
| JANUS [70] | F | FS | mem,sem | Sta, Sch, Cov | |
| Hydra [16] | F | FS | sem | Sta, Sch, Cov | |
| Razzer [7] | H | conc. | conc | Con, Dir, Cov, Tnt | |
| SegFuzz [8] | F | conc. | conc | Con, Cov | |
| Snowboard [9] | F | conc. | conc | Con, Sch, Dep | |
| SyzDirect [73] | F | directed | mem,logic | Dir, Dep, Inp, Sch, Tnt | |
| SyzRisk [108] | F | regression | mem,logic | Dir, Sch, Cov | |
| SyzGen [66] | F | spec-gen | n/a | Inp, Dep, Sym, Cov, Min | |
| KSG [67] | F | spec-gen | n/a | Inp, Sym, Tnt | |
| SyzDescribe [68] | S | spec-gen | n/a | Inp, Dep, Tnt | |
| SyzGen++ [76] | F | spec-gen | n/a | Dep, Inp, Sym, Cov | |
| FuzzNG [77] | F | core | mem | Inp, Cov | |
| KernelGPT [69] | F | spec-gen | n/a | Inp, Dep, ML | ★ |
| Enterprise [150] | E | generic | mem,conc,logic | Cov | |
| Directed-Ind. [149] | E | directed | mem,logic | Dir, Inp, Sch, Cov | |
| *Static and hybrid discovery* | | | | | |
| DR. CHECKER [18] | S | driver | mem,logic | Tnt | |
| K-Miner [19] | S | core | mem | Tnt | |
| CRIX [20] | S | generic | logic (chk) | Min, Tnt | |
| LRSan [21] | S | generic | logic (chk) | Min, Tnt | |
| UBITect [22] | S | generic | mem (UBI) | Sym, Tnt, Typ | |
| K-MELD [23] | S | modules | mem (leak) | Min, Typ, Tnt, Sym | |
| Goshawk [24] | S | generic | mem | Min, Typ, Tnt, Sym, ML | |
| DCUAF [10] | S | driver | conc (UAF) | Tnt, Min | |
| DEADLINE [15] | S | generic | conc | Sym, Tnt | |
| VulDeePecker [153] | S | generic | mem | ML, Tnt | |
| CheQ [36] | S | generic | logic (chk) | Min, Tnt | |
| Uninit-Bin [39] | S | binary | mem (uninit) | Sym, Tnt | |
| SUTURE [78] | S | generic | taint | Tnt, Sym | |
| Kconfig [110] | S | config | logic | Sym | |
| KUBO [29] | S | generic | mem (UB) | Sym, Tnt | |
| DEPA [38] | S | generic | logic | Min, Tnt | |
| MANTA [32] | S | container | mem (acct) | Tnt | |
| IncreLux [109] | S | generic | mem (UBI) | Sym, Tnt | |
| PATA [28] | S | generic | mem,logic | Tnt, Typ, Sym | |
| UACatcher [14] | S | driver | conc (UAC) | Typ, Tnt, Sym | |
| Err-Spec [37] | S | generic | logic (err) | Min, Tnt, Sym | |
| LLift [30] | S | generic | mem (UBI) | Sym, ML | ★ |
| IMMI [31] | S | generic | mem | Min, Typ, Tnt, ML | ★ |
| BugLens [79] | S | generic | taint | Tnt, Sym, ML | ★ |
| UAFX [80] | S | generic | mem (UAF) | Typ, Tnt, Sym | |
| SCAD [98] | H | network | sem | Sym, Tnt | |
| CPG [152] | S | generic | mem,logic | Min, Tnt | |
| Coccinelle [27] | S | generic | logic | Min | |
| KNighter [26] | S | generic | logic | Min, ML, Sym | ★ |

# Appendix D.
# Meta-Review

The following meta-review was prepared by the program committee for the 2027 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

## D.1. Summary

This paper presents a systematization of knowledge on the operating-system kernel bug lifecycle, spanning discovery, triage, patch generation, validation, and integration, primarily in the context of Linux. The authors introduce an "automation gradient" to characterize how automation maturity declines across these stages, support this framing with an empirical analysis of thousands of fixed kernel bugs, and evaluate a dozen LLM-based patch-generation approaches.

## D.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field

## D.3. Reasons for Acceptance

1) The paper provides a valuable step forward in an established field. Prior work has separately studied bug discovery, patch generation, and vulnerability lifetimes in the kernel, but this paper offers a unifying end-to-end perspective and explains, rather than merely observes, why automation succeeds at discovery but degrades in later stages, tying this explicitly to kernel-specific challenges such as concurrency, implicit invariants, weak correctness oracles, and hardware dependence.
2) The paper is grounded in substantial empirical analysis, examining thousands of fixed syzbot bugs to quantify the crash-to-patch gap and evaluating multiple LLM-based repair approaches against real kernel crashes, giving the systematization empirical weight beyond a literature summary.