pPatch: Automated Vulnerability Unpatching

TIANYI JING*, Huazhong University of Science and Technology, China
PENGYU DING*, Huazhong University of Science and Technology, China
MENG XU, University of Waterloo, Canada
YINHAO HU, Huazhong University of Science and Technology, China and Zhongguancun Laboratory, China
ZHENG YU, Northwestern University, USA
DONGLIANG MU, Huazhong University of Science and Technology, China

Unpatching, the process of reverting security patches to reintroduce historical vulnerabilities into newer software versions, is valuable for creating realistic benchmarks to evaluate security analysis tools. However, this process is challenging due to code evolution, leading to context conflicts, compilation errors, or untriggerable issues. In fact, 61.25% of Linux kernel security patches we examined cannot be trivially reverted to recent versions. To address this, we propose pPatch, an automated framework designed to systematically unpatch security vulnerabilities and generate vulnerability benchmark. pPatch overcomes the limitations of naive reversion by employing a novel approach that progressively consults conflicting commits to identify and integrate necessary code changes, aiming for minimal modifications to preserve program semantics while successfully re-exposing the original vulnerability and minimizing unintended side effects. Then pPatch unpatches 614 historic kernel vulnerabilities from Linux kernel v6.6 and v6.12, resulting in 371 and 353 successfully unpatched vulnerabilities with manual analysis. Based on the validation with Proof-of-Concept (PoC) and automated fuzzing, we constructed KVulnBench with 187 verified vulnerabilities, the first automatically generated and verified high-quality vulnerability dataset specifically for the Linux kernel. Using KVulnBench we evaluated the performance of state-of-the-art kernel security tools (e.g., syzkaller), demonstrating that KVulnBench provides a valuable resource for realistically assessing kernel security tools.

CCS Concepts: • **Security and privacy** → *Software and application security*; Software security engineering;

Additional Key Words and Phrases: Vulnerability Unpatching, Fuzzing Benchmark, Kernel Fuzzing

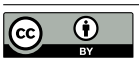
ACM Reference Format:
Tianyi Jing, Pengyu Ding, Meng Xu, Yinhao Hu, Zheng Yu, and Dongliang Mu. 2026. pPatch: Automated Vulnerability Unpatching. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE194 (July 2026), 25 pages. https://doi.org/10.1145/3808201

1 Introduction

Unpatching refers to the deliberate process of reverting previously applied patches and subsequently integrating these reversions into more recent software versions. In a security context, if a patch fixes a vulnerability, undoing the patch correctly means reinstating the vulnerability in the new version

*Both authors contributed equally to this research.

Authors' Contact Information: Tianyi Jing, Huazhong University of Science and Technology, Wuhan, China, linux@hust.edu.cn; Pengyu Ding, Huazhong University of Science and Technology, Wuhan, China, pengyu_ding@hust.edu.cn; Meng Xu, University of Waterloo, Waterloo, Canada, meng.xu.cs@uwaterloo.ca; Yinhao Hu, Huazhong University of Science and Technology, Wuhan, China and Zhongguancun Laboratory, Beijing, China, dddddd@hust.edu.cn; Zheng Yu, Northwestern University, Evanston, USA, zheng.yu@northwestern.edu; Dongliang Mu, Huazhong University of Science and Technology, Wuhan, China, dzm91@hust.edu.cn.



ACM 2994-970X/2026/7-ARTFSE194
https://doi.org/10.1145/3808201

while maintaining the functionality. Unpatching bears a conceptual similarity to backporting—a common practice in software engineering where fixes or improvements from a newer version are adapted for older versions [35, 47]. However, while backporting aims to enhance the security and functionality of legacy systems by applying modern patches, unpatching intentionally weakens a more recent software version by reverting the patches and recreating the original flaws.

While uncommon in software development, unpatching is particularly useful for creating realistic benchmarks of vulnerabilities that reflect real-world bug patterns [12, 48] or more precisely, historical bugs in the same codebase. By reviving vulnerabilities that once existed in the same software, researchers and developers can rigorously test the effectiveness of security tools e.g., fuzzers, static analyzers, and even automated program repair (APR) solutions. In essence, unpatching enables the controlled and systematic evaluation of security tools against authentic vulnerability scenarios.

However, unpatching is not a trivial "`git revert`". In fact, 61.25% of security patches we tested in the Linux kernel cannot be trivially reverted in a recent version (*e.g.,* v6.6.70) for a diverse set of reasons causing conflicting hunks, compilation errors. Furthermore, while similar challenges might also present in backporting, we discover that heuristics tuned for backporting cannot be readily applied in unpatching: backporting tends to favor deletions in the commit while porting an old, reverted commit forward tends to require additions from other commits (details in §2.3 and §3).

In this paper, we propose PPatch, an end-to-end unpatching tool specially designed to revert security patches in order to construct customized, realistic, and scalable vulnerability benchmarks in an automated pipeline. PPatch takes as input 1) a set of security patches to be reverted, 2) proof-of-concept (PoC) inputs that trigger these patched vulnerabilities, and 3) a target version where vulnerabilities should be reinstated. PPatch then follows a fully automated pipeline to unpatch each vulnerability, verify the "correctness" of each reversion, and integrate all unpatched and verified reversions in the designated version of the codebase.

PPatch tackles cases when trivial `git revert` or the GNUPatch utility fails, and the most prominent insight is to progressively consult conflicting commits (i.e., diffs that invalidate code contexts where the original patch was applied) for relevant code blocks to bring into the unpatch. PPatch is intentionally conservative in unpatch building, aiming to re-expose the original vulnerability *with minimal code changes* in order to minimize potential side-effects (e.g., distorting functionality or introducing unintended bugs). Once an unpatch candidate is produced, beyond manual checking, we also instruct PPatch to follow a multi-step validation process to check whether the unpatch can trigger the introduced vulnerability and gain confidence on the absence of unintended behaviors.

To showcase the efficacy of PPatch, we sourced 614 vulnerabilities (Details about vulnerability selection in §4) from Syzbot [38] that once existed in the Linux kernel and were subsequently patched across v4.13 to v6.8, and used PPatch to unpatch them in kernel v6.6.70 and v6.12.16 (the latest two up-to-date *longterm release* at the time of writing). Out of the 614 vulnerabilities, PPatch successfully produced 503 plausible unpatches, out of which 395 compiles, 387 passes Linux Test Project (LTP) tests [32], 132 can be triggered using the vanilla PoC, and 55 can be triggered with additional tweaks based on the PoCs. Repeating the same experiment on kernel v6.12.16 yield similar results—showcasing the applicability of using PPatch to construct vulnerability benchmarks across evolving versions of the Linux kernel.

Based on the 187 unpatched vulnerabilities on v6.6.70—for each of which we have a concrete PoC trigger—we construct a dataset, KVulnBench, by selecting the largest set of unpatches (157) that can be merged together, while also balancing the diversity of vulnerability types. KVulnBench is intended for benchmarking kernel-oriented security tools. We benchmarked state-of-the-practice kernel-oriented security tools with KVulnBench. While three static analyzers only discover 9 vulnerabilities in KVulnBench in total, kernel fuzzers show a clear advantage. Syzkaller [42]

discovers 37 of vulnerabilities in KVulnBench, almost three times more than Healer [37]. Time-to-vulnerability analysis showed Healer converged faster (avg. <6h) on simpler bugs, while Syzkaller's longer exploration (avg. >18h) proved necessary for more complex flaw discovery. In terms of directed fuzzing, due to resource limitations, we randomly sampled a subset of 30 vulnerabilities from KVulnBench and SyzDirect [41] was able to detect 14 of them, which demonstrated strong performance in the directed exploration.

To the best of our knowledge, KVulnBench is the first vulnerability benchmark tailored to the Linux kernel. KVulnBench is unique in methodology that it is constructed via an automated pipeline in contrast with MAGMA's manual approach [12]. Therefore, KVulnBench can be regularly updated in an automatic way as the kernel evolves. Compared with the vulnerability synthesis approach proposed in FixReverter [48], KVulnBench only contains authentic bugs that had historically existed in the kernel where each bug is guaranteed to be triggerable with a PoC.

Summary: this paper makes the following contributions:

- We propose pPatch, an automated unpatching framework that can re-instate historical vulnerabilities into more recent program versions with a reasonable level of effectiveness.
- Merging only unpatched vulnerabilities that are verified with concrete PoCs, we produce a vulnerability benchmark on Linux kernel v6.6.70, KVulnBench, consisting of 187 vulnerabilities across 74 subsystems. A similar benchmark for the v6.12.16 kernel is also available.
- We benchmark several state-of-the-art kernel fuzzers and static analyzers using KVulnBench, highlighting both their differences and limitations, and we hope KVulnBench can benefit future kernel-oriented security tools in serving as an objective vulnerability benchmark.

2 Motivation and Challenges

In this section, we describe the motivation of this research: automated generation of customizable and scalable vulnerability benchmarks, its challenges, and why backporting tools, which also tries to apply git diffs in different contexts, are not well suited for unpatching.

2.1 Vulnerability Benchmarks

A vulnerability benchmark is a set of known vulnerabilities *deliberately* chosen, crafted, and planted in target programs. Such a benchmark typically serves as a common ground to evaluate the effectiveness of various security tools, *e.g.,* fuzzers, static analyzers, and automated program repair (APR) tools. Exemplary vulnerability benchmarks such as MAGMA [12] and FixReverter [48] are now valuable community resources and have been widely used in benchmarking the effectiveness of fuzzers and thus advancing the field of research.

MAGMA is a manually curated ground-truth vulnerability benchmark consisting of real-world userspace programs, i.e., for each vulnerability included, its precise location, trigger conditions, and root cause are known *a priori*, derived from analyzing historical bug reports and their corresponding patches. To re-introduce a bug, its patch is reverted and manually forward-ported into a modern version of the target program. MAGMA has been used to benchmark a fleet of fuzzers, providing tangible insights in fuzzing research, *e.g.,* why crash counts are often misleading and how randomness affects fuzzer performance [1, 17].

MAGMA's strengths lie in the high-fidelity replication of realistic bugs, i.e., all planted bugs have occurred in the target codebase. However, its reliance on manual analysis and patch reversion is labor-intensive, limiting its scalability and the timeliness (i.e., incorporating new vulnerabilities)

posing a challenge for rapidly evolving codebases like the Linux kernel. These characteristics highlight a need for vulnerability benchmarks that can automatically adapt, maintain, and scale.

FixReverter is an automated bug injection framework that reintroduces *synthetic* vulnerabilities by reversing real-world bugfix patterns summarized from real-world CVE patches. Based on an analysis of 814 CVEs, it identifies three common fix patterns—*conditional-abort (ABORT)*, *conditional-execute (EXEC)*, and *conditional-assign (ASSIGN)*—and applies grammar-based syntactic matching to locate code segments where a synthetic bug can be introduced. Static reachability and dependence analyses are then used to ensure that the reversed patches can realistically lead to crashes, improving the reliability of injected bugs.

While FixReverter represents a significant step towards automated and realistic bug injection, it inherits the limitations of static analysis [48] in checking semantic conditions (e.g., reachability from entry points, data dependence to potential crash sites). Factors such as unsound handling of complex language features (e.g., function pointers) and incomplete modeling of library function interactions can lead to inaccuracies in analysis results. Consequently, FixReverter cannot guarantee that every injected bug is semantically triggerable or leads to a crash under dynamic execution; instead, it only suggests a likelihood based on the static checks performed. Moreover, relying on a small, predefined set of vulnerability patterns restricts the diversity of injected bugs, potentially failing to capture the broad spectrum and evolving nature of real-world vulnerabilities [11]. Similarly, Encarsia [4] also employs fixed-pattern bug injection to construct fuzzing evaluation benchmarks, with its focus placed on CPU hardware designs rather than software.

A new methodology to curate vulnerability benchmark. While MAGMA and FixReverter pioneer valuable approaches to vulnerability benchmark curation, building new vulnerability benchmarks, particularly for the complexities of the Linux kernel, still hit some of their shortcomings. Ideally, we need a tool that combines the advantages of MAGMA and FixReverter—a tool that can recreate realistic and verifiable vulnerabilities, especially those that once appeared in the same codebase without relying on heuristic vulnerability patterns (like MAGMA), but with minimal manual effort and full automation once the set of vulnerabilities to recreate is determined (like FixReverter). In this way, the vulnerability benchmark is both customizable—allowing researchers and practitioners to tailor the benchmark to specific testing and evaluation needs; and self-refreshing—allowing the benchmark itself to dynamically adapt to new codebase versions, reflecting emerging threats and newly discovered vulnerabilities.

Alternative: using an old version with many known bugs as a benchmark? We performed an empirical analysis of Linux kernel CVEs using detailed data in the kernel community's `vuln.git` repository [25]. Our calculations show that the Linux kernel v6.6.70 and v6.12.16 contain 251 and 24 vulnerabilities, respectively. Although these numbers may seem substantial, these CVEs exhibit low reproducibility rates (4.5%–43.8%), primarily due to insufficient information [29]. This issue is exacerbated by the Linux kernel's silent bug policy, making reproducing vulnerabilities more challenging. Another reason for not using an old version with known bugs as a benchmark is such a strategy will rule out the possibilities of vulnerability customization.

2.2 Challenges in the Unpatching

Intuitively, a decent vulnerability benchmark can be built in a fully automated way by programmatically reinstating vulnerabilities that historically exist in the same software codebase. This is especially true if, for a known vulnerability (e.g., as identified by a bug report or a CVE number), we know the precise patch that fixes the vulnerability. This patch, often in the form of a commit in the version control system, is called the original fix commit. Restoring the vulnerability is conceptually

```
--- a/drivers/net/usb/qmi_wwan.c
+++ b/drivers/net/usb/qmi_wwan.c
@@ -1482,7 +1482,7 @@ static int qmi_wwan_probe(struct
  ↪  usb_interface *intf,
     * different. Ignore the current interface if
     * equals the number for the diag interface (two).
     */
-    info = (void *)&id->driver_info;
+    info = (void *)id->driver_info;

     if (info->data & QMI_WWAN_QUIRK_QUECTEL_DYNCFG) {
          if (desc->bNumEndpoints == 2)
```

```
static int qmi_wwan_probe(struct usb_interface *intf,
                          const struct usb_device_id *prod)
{
    ...
     * different. Ignore the current interface if
     * equals the number for the diag interface (two).
     */
    if (desc->bNumEndpoints == 2)
        return -ENODEV;

    return usbnet_probe(intf, id);
}
```

Fig. 1. Linux kernel patch@904d88 and the code snippet of kernel v6.6.70 where patch@904d88d743b0 was previously applied.

```
--- a/drivers/net/usb/qmi_wwan.c
+++ b/drivers/net/usb/qmi_wwan.c
@@ -1491,12 +1483,8 @@ static int qmi_wwan_probe(struct
  ↪  usb_interface *intf,
     * different. Ignore the current interface if
     * equals the number for the diag interface (two).
     */
-    info = (void *)id->driver_info;
-
-    if (info->data & QMI_WWAN_QUIRK_QUECTEL_DYNCFG) {
-        if (desc->bNumEndpoints == 2)
-            return -ENODEV;
-    }
+    if (desc->bNumEndpoints == 2)
+        return -ENODEV;

     return usbnet_probe(intf, id);
 }
```

Fig. 2. The essential hunk needed in the patch revision. It removes the reference of `info` and blocks the unpatching of `patch@904d88`.

```
--- a/drivers/net/usb/qmi_wwan.c
+++ b/drivers/net/usb/qmi_wwan.c
@@ -61,7 +61,6 @@ enum qmi_wwan_flags {
 enum qmi_wwan_quirks {
     QMI_WWAN_QUIRK_DTR = 1 << 0,
-    QMI_WWAN_QUIRK_QUECTEL_DYNCFG = 1 << 1,
 };

 struct qmimux_hdr {
@@ -1455,7 +1448,6 @@ static int qmi_wwan_probe(
 {
     struct usb_device_id *id = (struct usb_device_id *)prod;
     struct usb_interface_descriptor *desc =
    ↪  &intf->cur_altsetting->desc;
-    const struct driver_info *info;

     /* Workaround to enable dynamic IDs.  This disables
      * blacklisting functionality.  Which, if required, can
```

Fig. 3. Partial hunks added to fix compilation errors, which contain the definition of `info` and `QMI_WWAN_QUIRK_QUECTEL_DYNCFG`.

as simple as first reverting the fix commit and subsequently adapt and apply hunks (continuous code blocks) in the reversion up to a recent commit.

Figure 1 shows the commit that fixes syzbot-b68605 [40]. This commit addresses a type-casting bug that, once triggered by a PoC input, can cause an out-of-bound memory access. Specifically, variable `info` was designed to store the value of `id->driver_info`, rather than its address. This fix is concise, as it simply removes the address-of operation (&) before `id->driver_info`. The original fix commit lands in Linux kernel v5.2.0-rc5. To re-introduce this vulnerability in a recent kernel version (*e.g.*, v6.6.70), we need to "peel off" this patch by reverting the changes in the v6.6.70 codebase. Unfortunately, with all code changes between the old version and v6.6.70, unpatching this commit is not trivial.

Challenge 1: Fail to reverse all patch hunks. At first, we manually utilize GNUPATCH to revert this patch. According to the output, patch@904d88 [20] fails to be reverted despite its simplicity. The working principle behind GNUPATCH is that it first checks whether the line numbers in the patch match the content to be modified at the specified location. If the patch does not include line numbers or the current location does not match the content to be modified, GNUPATCH performs a context-matching search. If the original patch location cannot be determined even with this method, GNUPATCH would declare the revert operation as failed.

After checking the code of Linux kernel v6.6.70 in Figure 1, the reason for this failure is that function `qmi_wwan_probe` makes some changes to Line 8~10 in the historic commits between the original fix commit and the target program. We refer to these historic commits that modified the locations of the original fix commit as **conflict commits**, and the specific hunks within these

```
drivers/net/usb/qmi_wwan.c: In function ‘qmi_wwan_probe’:
drivers/net/usb/qmi_wwan.c:1553:9: error: ‘info’ undeclared (first use in this function)
 1553 |         info = (void *)&id->driver_info;
      |         ^~~~
drivers/net/usb/qmi_wwan.c:1553:9: note: each undeclared identifier is reported only once for each function it
drivers/net/usb/qmi_wwan.c:1555:26: error: ‘QMI_WWAN_QUIRK_QUECTEL_DYNCFG’ undeclared (first use in this
 1555 |         if (info->data & QMI_WWAN_QUIRK_QUECTEL_DYNCFG) {
      |                          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Fig. 4. The compilation error message reported by GCC after reversing conflict hunks.

conflict commits that introduced modifications are termed **conflict hunks**. By identifying the conflict commits and hunks, we can understand what happened to the failed hunks in the git history, thereby enabling the original fix commit to be successfully reversed.

Using `git log -p`, we attempt to search backward through the commit history of the file modified in patch@904d88 to identify conflict commits that altered the apply location. Through this method, the conflict commit for patch@904d88 is identified as commit@00516d [21], shown in Figure 2. The Hunk 6 deletes the assignment operation to the `info` variable, which corresponds to the location of the original fix commit. After this hunk is reversed, patch@904d88 can be successfully reverted.

Challenge 2: Compilation errors. Ideally, after resolving the conflict hunks and reversing the patch@904d88, the unpatched program should compile successfully. However, this is usually not the case: when compiling the unpatched program using the same compilation options as the target program, the compiler produces errors, as shown in Figure 4. The error message indicates that the definitions of the variables `info` and `QMI_WWAN_QUIRK_QUECTEL_DYNCFG` cannot be found.

In Challenge 1, while attempting to revert patch@904d88, we initially reverted only one hunk originating from conflicting commit@00516d. However, reversion of this hunk brings back variables `info` and `QMI_WWAN_QUIRK_QUECTEL_DYNCFG`. In commit@00516d, definitions and references associated with the two variables had been removed simultaneously. But our initial reversion restored the references without restoring the corresponding definitions, resulting in a compilation failure.

This situation occurs due to our conservative strategy on the hunk selection. We opted to selectively identify and apply only necessary hunks from the conflicting patch to minimize the impact on the existing codebase and ensure the correctness of reintroduced code changes. We manually inspected commit@00516d to locate hunks containing definitions or references related to `info` and `QMI_WWAN_QUIRK_QUECTEL_DYNCFG`. This inspection revealed that some hunks were relevant. For example, Hunk 1 removed `QMI_WWAN_QUIRK_QUECTEL_DYNCFG` from the `enum qmi_wwan_quirks` and Hunk 5 removed the definition of `info` variable, as shown in Figure 3.

Upon sequentially reverting these three identified hunks from commit@00516d, the original target patch@904d88 could be successfully reverted, and the unpatched program subsequently compiled without compilation errors.

Challenge 3: Triggering the re-exposed vulnerability (and not something else). Upon successful compilation of the unpatched kernel, we deploy it in QEMU and execute the original PoC to verify the introduced vulnerability. Unfortunately, the vanilla PoC failed to trigger the introduced vulnerability. Manual debugging revealed that structural changes in the kernel's evolution had invalidated a critical array index within the PoC. By correcting this index to align with the target version, we successfully reproduced the desired crash, confirming the vulnerability's presence.

However, this isn't the end of the process for benchmark generation. We still need to answer a remaining question: how do we know that the unpatch does not introduce side effects, such as distorting functionalities or introducing new bugs? And more importantly, besides manual verification, are there automated ways to boost our confidence on the correctness of the unpatch?

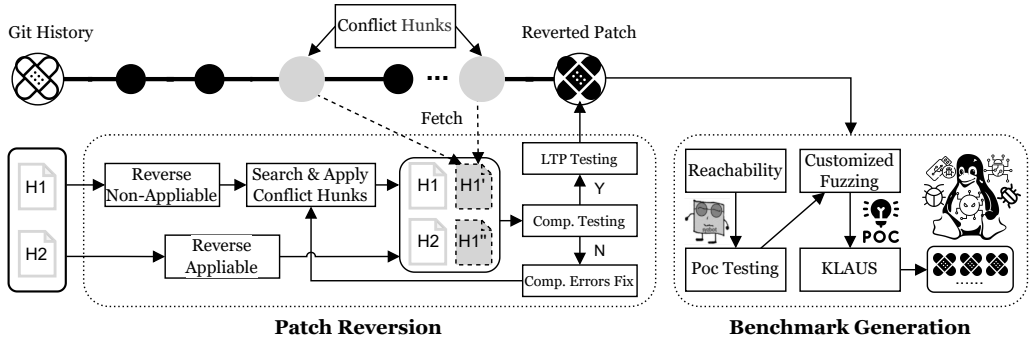


Fig. 5. **The workflow of pPatch**. **Hunk H1′** and **Hunk H1″** are added to resolve mismatch context and fix compilation errors of **Hunk H1**, respectively.

From Challenges to Design Requirements. The qmi_wwan example (patch@904d88) demonstrates that automated unpatching requires more than patch application. **First**, code evolution breaks the original patch context, so an unpatching system must recover the missing context from conflict commits while keeping edits minimal; **Second**, even after the reverted hunk becomes applicable, conservative hunk selection can leave semantically coupled declarations or definitions behind, causing compilation failures; **Third**, successful compilation alone is insufficient: the reintroduced vulnerability must still be triggerable in the target version, and the unpatch should avoid observable regressions. These observations motivate pPatch's design: (i) syntax reconciliation for context recovery, (ii) semantic error repair for compiler-guided recovery of missing symbols, and (iii) a multi-phase validation approach for re-triggerability and patch correctness.

2.3 Backporting vs. Unpatching

While §2.2 shows how the challenges of re-instating a historical vulnerability via unpatching can be resolved manually, in this paper, we seek an automated way to achieve this and our first attempt is to re-purpose backporting tools [35, 47]. Backporting refers to migrating patches from newer versions to older versions. Its core challenges lie in: 1) locating the corresponding position of the patch in the older version, 2) transforming the patch to be compatible with the older codebase, and 3) checking that the backported patch does not introduce new security holes. These challenges, at first glance, resonate well with the challenges unpatching faces.

However, we soon realized that the fundamental difference in diff porting direction invalidates some key heuristics used in state-of-the-art backporting tools, such as Fixmorph [35] and TSBPORT [47]. For example, Fixmorph and TSBPORT rely on predefined rules to perform patch localization and transformation at both syntactic and semantic levels. However, their rules are specific to the scenario where a later patch needs to be applied to an older commit, making them unsuitable for direct application in unpatching tasks (which reverts an old patch in a recent commit). As illustrated in Figure 1, we attempted to use Fixmorph and TSBPORT to perform unpatching. Specifically, we revert patch@904d88 shown in Figure 1 and try to migrate the reversion to kernel v6.6.70. However, as shown in Figure 1, Line 7 in Figure 1 does not exist in v6.6.70. Hence, TSBPORT chooses not to migrate this patch. This behavior inherently follows the underlying logic of backporting: since its primary goal is to fix vulnerabilities, the absence of the vulnerable code implies that no patching is needed. It is precisely this implicit rule that makes existing backporting tools inapplicable to unpatching scenarios. Similarly, Fixmorph also fails to migrate the reversion

because it cannot locate the corresponding code position in the target version. Furthermore, we conduct a comprehensive comparative evaluation of pPatch against Fixmorph and TSBPORT on a unified dataset in §4.1, and present an in-depth case study of evaluation results in the Appendix A.

3 pPatch Core: Unpatching for A Single Vulnerability

Reverting the fix for a single vulnerability is the core of pPatch and is the foundation for building a many-vulnerability benchmark. In this section, we describe the design details for this core task, with the running example patch@904d88 in §2, throughout this section.

3.1 pPatch Overview

Design Overview and Challenge Mapping. Figure 5 summarizes pPatch with two stages. pPatch adheres to a conservative design principle: introducing minimal code changes when reverting the original patch, and this guides every aspect of its architecture and implementation. The design follows directly from the challenges in §2.2.

Stage I: Patch Reversion. To address failed reversion caused by code evolution (**Challenge 1**), pPatch performs *Syntax Reconciliation*, which detects failed hunks in the reverted patch and replays only overlapping conflict hunks from the intervening history. To address compilation failures caused by semantically coupled changes (**Challenge 2**), pPatch performs *Semantic Error Repair*, which extracts missing symbols from compiler diagnostics and retrieves additional hunks that restore the required declarations/definitions.

Algorithm 1: Conflict Commits Resolution.

Input: Target program P, target commit C_{tar}, and commit to be reverted C_{rev}
Output: Reverted program P_{rev}

Function `ResolveConflictCommits(`P, C_{tar}, C_{rev}`):`
  $p_{\text{orig}} \leftarrow \texttt{diff}(C_{\text{rev}}, C_{\text{rev}}^{\sim 1})$;
  $H_{\text{fail}} \leftarrow \emptyset$, $P' \leftarrow P$;
  foreach *hunk* $\in p_{orig}$ **do**
    if *hunk cannot be applied to* P' **then**
      $H_{\text{fail}} \leftarrow H_{\text{fail}} \cup \{\text{hunk}\}$;
    else
      $P' \leftarrow \texttt{apply}(P', \text{hunk})$;
  for *commit* ***from*** C_{tar} ***to*** C_{rev} **do**
    $p_{\text{commit}} \leftarrow \texttt{diff}(\text{commit}, \text{commit}^{\sim 1})$;
    foreach *hunk* $\in p_{commit}$ **do**
      foreach $h_{fail} \in H_{fail}$ **do**
        if *hunk overlaps with* h_{fail} **then**
          $P' \leftarrow \texttt{apply}(P', \text{hunk})$;
          break;
  $P_{\text{rev}} \leftarrow \texttt{apply}(P', H_{\text{fail}})$;
  return P_{rev};

Stage II: Benchmark Generation. To address triggerability drift and ensure correctness (**Challenge 3**), pPatch performs *Vulnerability Validation*, a multi-phase process approach that confirms the introduced vulnerability can indeed be triggered. Validation begins with the reachability analysis, followed by PoC testing to check if the original proof-of-concept can trigger crashes in the unpatched program. If both fail, a customized fuzzer is employed with a focus on mutating the original PoC to find an input that can trigger the corresponding vulnerability. To reduce the risk of unintended behaviors, pPatch additionally performs patch testing (*e.g.,* LTP and KLAUS) and discards candidates that introduce new failures or unrelated crashes. Furthermore, pPatch conducts *Reverted Patch Merging* to integrate multiple reverted patches into a single program version by resolving conflicts between patches that modify the same files. It constructs a conflict graph over patches, ranks them by conflict degree and other attributes using a greedy prioritization strategy, and finally selects minimally conflicting patches while pruning incompatible alternatives to maximize benchmark coverage.

3.2 Syntax Reconciliation

pPatch identifies conflict commits and merges necessary parts of these conflict commits into the reverted patch, generating a syntactically valid patch. In our running example (patch@904d88), pPatch identifies Hunk 6 of `commit@00516d` as a conflict hunk because it overlaps the failed reverted hunk in `qmi_wwan_probe`, thereby recovering the missing context for applying the reversion. Specifically, the final patch should be able to be applied to the target program by GNUPatch. As *Challenge 1* we stated in §2, the original reverted patch@904d88 may not be directly applicable to the target program because some hunks of patch@904d88 may conflict with the target program. To resolve these conflicts, Algorithm 1 presents a systematic approach where GNUPatch first identifies the failed hunks during the application of patch@904d88, and then locates and applies the necessary intermediate changes to enable successful patch reversion.

The Algorithm 1 operates in three distinct phases. Initially, it attempts to apply each hunk from the original patch (p_{orig}) to the current program version, identifying hunks that fail due to code modifications that occurred between two commits. These failed hunks are collected in H_{fail} while successfully applied hunks modify the intermediate program state P'. This identification step is crucial as it pinpoints exactly which parts of the original patch require special handling. Since patch@904d88 contains only one hunk, the resulting set H_{fail} ultimately includes only that hunk.

In the second phase, the algorithm examines all commits between the target commit C_{tar}, usually the current version, and the commit to be reverted C_{rev}. For each intermediate commit, it extracts the corresponding patch, identifies the specific lines modified by each hunk, and examines whether any of these hunks overlap, in terms of modified lines, with the previously identified failed hunks. In Algorithm 1 (line 9), when iterating over commits, the line ranges of hunks in H_{fail} are updated according to the insertions and deletions introduced by each commit, so that they are always aligned with the current version of the file. This allows pPatch to determine hunk overlaps by line-range intersection. During the traversal from patch@904d88 to the current version, the algorithm detects such an overlap in commit@00516d, where Hunk 6 modifies lines affected by the original patch. Upon this detection, the algorithm applies the corresponding intermediate hunk to P', effectively reconstructing the contextual changes necessary for the failed hunks to be applied correctly. This process ensures that all code dependencies and context modifications that occurred between commits are properly accounted for, resulting in a program state that can accept the previously failed hunks. Finally, as shown in Algorithm 1, all remaining failed hunks are applied to produce the reverted program P_{rev}.

3.3 Semantic Error Repair

By identifying the conflict hunks that the original patch depends on, pPatch successfully generates a syntactically valid reverted program. However, resolving syntax conflicts alone is insufficient. As the example in *Challenge 2* shown in §2 demonstrates, we initially selected only the conflict hunks required to revert patch@904d88. However, to resolve the compilation errors, we revisited commit@00516d in *Challenge 2* and added the additional hunks containing the definition of the variable `info`. Through investigation, we found that symbol-related modifications are often scattered across multiple hunks, and that missing symbols are the primary cause of compilation failures after conflict resolution. These missing symbols occur because not all hunks related to the variables reported in the compilation errors were selected, resulting in the absence of definitions or declarations for these symbols in the reverted program.

Based on this observation, pPatch adopts a systematic strategy to repair semantic errors. To minimize the number of hunks selected and added while ensuring that no hunks related to the variable are overlooked, pPatch extracts the variables mentioned in the compilation errors and

marks them as symbols. For instance, in the motivating example of patch@904d88, the compiler error indicates that the variable `info` is undefined, and PPATCH extracts `info` as a required symbol. All modifications added by the system originate from conflict commits. Therefore, the scope of adding these hunks also starts from conflict commits, searching for hunks containing the extracted symbols across all conflict commits, as shown in Algorithm 2. All added and deleted lines within each hunk are included in the search scope to cover all scenarios of variable additions, deletions, and modifications. During the traversal of git history, PPATCH searches for hunks containing these symbols in the commits corresponding to conflict commits and hunks, and denotes them as additional hunks. In the running example, this process identifies extra hunks in commit@00516d that provide the definition of `info`. The lines involved in these additional hunks are also flagged. If subsequent commits modify these additional hunks, they similarly become unrevertible. Therefore, similar to the previous operations, the hunks modifying these additional hunks are also flagged and sequentially reverted to eventually restore the original patch. If compilation errors on the reverted version persist, PPATCH repeats the process of selecting additional hunks. This iterative process terminates either when the maximum limit of three attempts is reached or when the compilation succeeds without errors.

3.4 Theoretical Analysis

Both Algorithm 1 and the Algorithm 2 adopt a similar iterative strategy that scans intermediate commits and matches relevant hunks. Let M denote the number of commits between the fix commit C_{rev} and the target version C_{tar}, K denote the average number of hunks per commit, and F denote the number of hunks that initially fail to apply. In the worst case, PPATCH needs to traverse all intermediate commits and compare each hunk with the set of failed hunks, resulting in a time complexity of $O(M{\cdot}K{\cdot}F)$. The space complexity is $O(F)$, since the algorithms only need to maintain metadata for the set of failed hunks and do not require storing the entire commit history in memory.

Algorithm 2: Semantic Error Repair via Adding Extra Hunks

Input: Target program P, target commit C_{tar}, commit to be reverted C_{rev}
Output: Semantically repaired reverted program P_{rev}

Function `RepairSemanticErrors`(P, C_{tar}, C_{rev})**:**
  $P_{\text{rev}} \leftarrow$ `ResolveConflictCommits`$(P, C_{\text{tar}}, C_{\text{rev}})$;
  $E \leftarrow$ `compile`(P_{rev});
  $i \leftarrow 0$;
  while $E \neq \emptyset$ ***and*** $i < 3$ **do**
    $i \leftarrow i + 1$;
    $S \leftarrow$ `extract_symbols`(E);
    $H_{\text{add}} \leftarrow \emptyset$;
    for *commit* ***from*** C_{tar} ***to*** C_{rev} **do**
      $p_{\text{commit}} \leftarrow$ `diff`$(\text{commit}, \text{commit}^{\sim 1})$;
      foreach *hunk* $\in p_{commit}$ **do**
        if *hunk contains any symbol in S* **then**
          $H_{\text{add}} \leftarrow H_{\text{add}} \cup \{\text{hunk}\}$;
          $P_{\text{rev}} \leftarrow$ `apply`$(P_{\text{rev}}, \text{hunk})$;
    $E \leftarrow$ `compile`(P_{rev});
  return P_{rev};

3.5 Implementation Highlights

PPATCH Core is implemented in Python with 1816 LoC. It automates patch reversion with a customized version of `whatthepatch` parser for unified patches. Meanwhile, it redesigns the patch application mechanism to incorporate with our conflicting hunk resolving mentioned in §3.2. To facilitate symbol location in the Linux kernel, we integrate `cscope`, leveraging its comprehensive C symbol identification capabilities. And a Python wrapper is developed to read symbol location information from the database generated by `cscope`.

Table 1. Comparison of pPatch and all evaluated tools in unpatching effectiveness. The table shows success rates for different tools across two Linux kernel versions (v6.6.70 and v6.12.16). **Reverse**, **Compile**, and **LTPTest** respectively represent the proportion of unpatches that pass each of these stages. The **Manual** column shows results of our manual analysis on the unpatches that pass LTPTest.

Kernel	Tool	Reverse	Compile	LTPTest	Manual
v6.6.70 *(include 609 bugs)*	GNUPatch	246 (40.39%)	244 (40.07%)	239 (39.57%)	236 (38.75%)
	GNUPatch-F 3	292 (47.95%)	281 (46.14%)	274 (45.36%)	270 (44.33%)
	TSBPORT	136 (22.50%)	106 (17.26%)	105 (17.24%)	70 (11.49%)
	FixMorph	254 (41.71%)	167 (27.42%)	163 (26.77%)	150 (24.63%)
	pPatch	503 (82.59%)	395 (64.86%)	387 (64.07%)	371 (60.92%)
v6.12.16 *(include 614 bugs)*	GNUPatch	223 (36.32%)	221 (35.99%)	216 (35.18%)	214 (34.85%)
	GNUPatch-F 3	269 (43.81%)	259 (42.18%)	252 (41.04%)	249 (40.55%)
	TSBPORT	145 (23.62%)	107 (17.43%)	106 (17.26%)	64 (10.42%)
	FixMorph	251 (40.88%)	162 (26.38%)	158 (25.73%)	145 (23.62%)
	pPatch	499 (81.27%)	386 (62.87%)	377 (61.40%)	353 (57.49%)

4 Evaluation of pPatch Core for Single Vulnerability Unpaching

In this section, we evaluate pPatch Core from two perspectives:

RQ1 How effective is pPatch in reverting **one** designated patch commit (and hence, re-introduce *one* vulnerability) in a target version of the codebase? This forms the core of pPatch.

RQ2 Can pPatch be applied to user-space programs? If so, how is the quality of unpatches created by pPatch compared with manually crafted unpatches?

Experiment setup. All experiments were conducted on two servers equipped with one AMD Ryzen 9 9950, 128GB of RAM and a total of 17.36 TB of SSD storage running Ubuntu 24.04 LTS.

Target programs and corresponding patches. The programs we selected to evaluate pPatch include: two most recent Linux kernel LTS versions (v6.6.70 and v6.12.16) and the same set of userspace programs selected by MAGMA [12]. We collect a total of 752 patches. For the Linux kernel, we curate 614 vulnerabilities reported by Syzbot [38] over the past three years. Specifically, starting from 3,012 non-duplicate Syzbot reports, we retain a case only if it satisfies three criteria: (i) the bug is detected by KASAN, (ii) an upstream fixing commit is provided, and (iii) a valid reproducer (PoC) is available and can be replayed to reliably trigger the crash. This process yields 614 kernel vulnerabilities. And the remaining userspace patches are drawn from MAGMA.

4.1 One-shot Unpatching (RQ1)

Lacking other automated unpatching tools for comparison, we selected the most commonly used patch tools, GNUPatch and its `-F 3` configuration which represents the most aggressive way to apply all hunks in a diff ignoring their surrounding contexts. In addition, to provide a broader comparison, we further evaluate existing backporting tools on the same unpatching tasks.

GNUPatch and GNUPatch-F 3. *-F* refers to the value of fuzz factor. Firstly, GNUPatch looks for a location where all context lines fit. If it cannot find such a location, it sets the fuzz factor to 1, ignoring the first and last lines of the context. The fuzz factor increases with the number of failed attempts, up to a maximum of 3, which corresponds to the length of the patch's context. By default,

GNUPATCH sets the fuzz factor to 2. The -F 3 option represents the most aggressive patch application strategy in GNUPATCH.

TSBPORT and FIXMORPH. Existing backporting tools are not designed for the unpatching scenario. In our evaluation, We use the latest available versions of TSBPORT and FIXMORPH. Without modifying their core implementations, we reconfigure their experimental settings so that the version containing the original patch is treated as the target version, and the current version is regarded as the version without prior backporting. This configuration enables TSBPORT and FIXMORPH to operate forward-porting. In particular, FIXMORPH cannot handle patches that involve source files other than C (e.g., header files). We excluded such patches from its evaluation and FIXMORPH processes 554 and 553 patches on Linux kernel versions 6.6.70 and 6.12.16, respectively.

Evaluation criteria. The unpatching performed by the evaluated tools may run into various levels of success, which we categorize as follows::

(1) **Reversable**: the patch (and associated code chunks) that fixes the vulnerability is reverted.
(2) **Compilable**: the unpatched version compiles error-free.
(3) **Functionally integral**: all functional tests (e.g., LTP) pass on the unpatched version.

To qualify as successful, an unpatch must preserve functional integrity. For those unpatches that satisfy this criterion, we additionally perform a manual review to ensure their correctness, verifying that the reverted changes do not include unrelated code modifications and can expose the intended vulnerability accurately without any observed unintended behaviors in our tests.

Results overview. Table 1 summarizes the results of PPATCH and all evaluated tools in unpatching each vulnerability in our benchmark (614 vulnerabilities for kernel v6.12.16 and 609 for version v6.6.70). We note the following observations based on the experiment results:

- Compared with the most aggressive form of GNUPATCH (i.e., `-F 3`), PPATCH produces 16.6% and 16.9% more unpatches respectively that are extensively checked to successfully re-expose the original vulnerability, with no additional regressions during LTP testing and manual triage.
- Existing backporting tools exhibit substantially lower success rates in forward porting. Across both kernel versions, the number of vulnerabilities successfully reverted by FIXMORPH and TSBPORT is significantly smaller than that achieved by PPATCH. Notably, their success rates are also lower than those of GNUPATCH `-F 3`, suggesting that backporting tools are not well-suited for forward porting.
- In a cross-kernel version comparison, PPATCH demonstrates slightly better performance on the older version (v6.6.70). By analyzing cases with different unpatching outcomes between the two versions, we found that unpatching the same vulnerability in the newer version involves more code commits, which introduces additional complexity for PPATCH to create a correct unpatch. A similar trend can also be observed for the evaluated backporting tools.

Failure Cases. While PPATCH shows a reasonable level of success in unpatching (nearly 60%), a subset of cases remains challenging. Our post-hoc examination reveals six recurring failure modes: ❶ context not found due to file/directory renaming; ❷ macro-expansion–induced symbol mismatches; ❸ excessive hunk reversal that removes unrelated lines; ❹ incomplete hunk reversal that omits semantically coupled changes (e.g., lock/unlock pairs); ❺ vulnerabilities fixed by multiple commits; ❻ erroneously labeled fix commits. We further analyzed the failure modes of the repurposed backporting tools. FIXMORPH rarely fails at the application stage because it generates fully migrated source files yielding inherently applicable diffs. Across the baselines, we observe four recurring failure categories: ❶ format issues (FIXMORPH and TSBPORT), where malformed or non-standard patch formatting breaks parsing or downstream tooling; required for generation; ❷

injected extra headers (TSBPORT), where additional header inclusions introduced during reversion lead to missing/incorrect dependencies in the target version; and ❸ syntactically incomplete patches (TSBPORT & FixMorph), where the produced diff is structurally partial (e.g., truncated hunks or unmatched markers), making it invalid for standard patch utilities. ❹ undefined symbol introduction (FixMorph), where the generated patch introduces variables or symbols that do not exist in the target version and whose definitions cannot be found in the current codebase, leading to compilation failures. Detailed qualitative analysis and representative case studies are deferred to Appendix A, including how pPatch resolves cases that these baselines fail to handle.

4.2 Extensibility (RQ2)

The capability of pPatch to revert security patches is not limited to the Linux kernel. Conceptually, any codebase with a complete commit history and labeled security patches can leverage pPatch to reintroduce historical vulnerabilities into a specified version, thereby constructing a vulnerability benchmark. However, practically, although the design of pPatch does not rely on any language-specific features, the implementation is tightly coupled with toolchains for C/C++. Therefore, in this part, we evaluate pPatch on unpatching C/C++ codebases of popular userspace programs.

Target selection. We select the same set of codebases used in the MAGMA benchmark [12] to evaluate how effective pPatch can be in unpatching userspace programs. This setup provides the most intuitive statistics to answer the following question: *how much manual unpatching effort could be reduced should pPatch be available when MAGMA was created or later updated?*

Quirks in the evaluation process. While MAGMA provides standardized manually crafted unpatches in its publicly available benchmark, the build script of underlying projects differs, so we added wrappers for different C/C++ build systems to simplify access to compiler outputs for pPatch. However, MAGMA still has two features that make the evaluation of pPatch less straightforward.

1) Lack of ground-truth PoCs for verifying unpatches. MAGMA does not provide a complete set of PoCs for all vulnerabilities it unpatched. Therefore, we only collected 60 PoCs out of 138 bugs in total. Moreover, two programs (`libsndfile` and `Lua`) are added after the publication MAGMA, and both programs do not have corresponding PoCs available. This implies that the number of triggerable unpatches might not be a reliable statistic to evaluate pPatch.

2) Availability of ground-truth unpatches. The fact that MAGMA provides ground-truth unpatches for all vulnerabilities enables manual unpatch verification as we can directly compare unpatches produced by pPatch with the ground-truth unpatches. This partially compensates for the fact that we don't have ground-truth PoCs to verify the triggerability of each unpatches.

Results overview. Table 2 summarizes the results of using pPatch for unpatching the same set of vulnerabilities over the same set of userspace programs identified that constitute the MAGMA benchmark. The results of pPatch are also compared with GNUPatch `-F 3` to measure the additional savings on manual effort in using pPatch. We note the following observations based on the experiment results:

- Among all targets, pPatch can produce the same or more *compilable* unpatches than GNUPatch.
- pPatch and GNUPatch produce the same number (more precisely, the same set) of unpatches that can be triggered using the vanilla PoCs collected. However, this does not necessarily imply that pPatch bears the same effectiveness as GNUPatch due to the fact that only 6% of planted bugs in MAGMA have PoCs available and all 8 PoCs correspond to bugs that can be revealed by both GNUPatch and pPatch.

Table 2. Comparison of pPatch and gnuPatch performance across MAGMA. **Reverse** and **Compile** respectively represent the proportion of unpatches that pass each of these two stages. **T** denotes the number of unpatched programs triggered using available PoCs. **R** represents the actual number of PoCs collected in the Compilable set of bugs. **Manual** indicates the number of unpatches produced by the **Tool** that have been manually verified to be logically consistent with the ground-truth unpatch.

Target	Tool	Reverse	Compile	T/R	Manual
libpng	Patch -F 3	4 (57.14%)	4 (57.14%)	3 / 3	4 (57.14%)
7 bugs 6 PoCs	pPatch	4 (57.14%)	4 (57.14%)	3 / 3	4 (57.14%)
libtiff	Patch -F 3	0 (00.00%)	0 (00.00%)	0 / 0	0 (00.00%)
14 bugs 7 PoCs	pPatch	12 (85.71%)	5 (35.71%)	0 / 0	5 (35.71%)
libxml2	Patch -F 3	13 (76.47%)	13 (76.47%)	7 / 7	13 (76.47%)
17 bugs 8 PoCs	pPatch	16 (94.12%)	15 (88.24%)	7 / 7	15 (88.24%)
openssl	Patch -F 3	8 (40.00%)	8 (40.00%)	1 / 1	8 (40.00%)
20 bugs 7 PoCs	pPatch	14 (70.00%)	11 (55.00%)	1 / 1	11 (55.00%)
poppler	Patch -F 3	0 (00.00%)	0 (00.00%)	0 / 0	0 (00.00%)
22 bugs 13 PoCs	pPatch	12 (54.55%)	3 (13.64%)	0 / 0	3 (13.64%)
sqlite3	Patch -F 3	10 (50.00%)	9 (45.00%)	6 / 6	9 (45.00%)
20 bugs 14 PoCs	pPatch	10 (50.00%)	9 (45.00%)	6 / 6	9 (45.00%)
php	Patch -F 3	7 (43.75%)	7 (43.75%)	1 / 1	7 (43.75%)
16 bugs 5 PoCs	pPatch	8 (50.00%)	7 (43.75%)	1 / 1	7 (43.75%)
libsndfile	Patch -F 3	8 (44.44%)	8 (44.44%)	N.A.	8 (44.44%)
18 bugs 0 PoCs	pPatch	16 (88.89%)	15 (83.33%)	N.A.	15 (83.33%)
lua	Patch -F 3	1 (25.00%)	1 (25.00%)	N.A.	1 (25.00%)
4 bugs 0 PoCs	pPatch	3 (75.00%)	3 (75.00%)	N.A.	3 (75.00%)
Total	Patch -F 3	51 (36.96%)	50 (36.23%)	18/18	50 (36.23%)
138 bugs 60 PoCs	pPatch	95 (68.84%)	72 (52.17%)	18/18	72 (52.17%)

- Manual comparison between the unpatches produced by gnuPatch, pPatch, and MAGMA ground-truth show that pPatch can correctly generate more than half of the unpatches (i.e., automatically construct half of MAGMA), with a considerable 15.9% improvements over gnuPatch. In fact, all compilable unpatches from pPatch are the same as the ground-truth unpatches.

Outliner cases. In `poppler` and `libtiff`, due to a large-scale code style refactoring, gnuPatch fails to locate the correct positions to apply the patches in all cases. In contrast, pPatch is able to effectively recover these changes. However, `sqlite3` does not use Git as its version control system, hence pPatch cannot bring back more vulnerabilities than the initial patch reversal effort (which is the same as gnuPatch).

5 pPatch Partner: Merging Unpatches for A Vulnerability Benchmark

Building on the ability to unpatch a single bug, in this section, we discuss how to construct a realistic and practical vulnerability benchmark that may consist of hundreds of individual unpatches.

5.1 Vulnerability Validation

Beyond the manual check that an unpatch is a semantic reverse of the corresponding bug fix, we need additional validation on the generated unpatches to build a practical vulnerability benchmark. More specifically, we need to provide a concrete proof-of-concept (PoC) input that can trigger the

Table 3. Proportion of unpatches across successive validation stages on two Linux kernel versions (v6.6.70 and v6.12.16). **Trigger**+ represents PoC tests combined with Syzkaller, and **Trigger**++ represents PoC tests combined with Syzkaller and KLAUS. The **No side-effect** column counts unpatches that introduce only the intended bug without additional issues when applied.

Kernel	Trigger	Trigger+	Trigger++	No side-effect
v6.6.70	132 (35.58%)	185 (49.87%)	187 (50.40%)	187 (50.40%)
v6.12.16	122 (34.56%)	165 (46.74%)	167 (47.31%)	167 (47.31%)

exposed vulnerability at runtime. While the intuitive solution is to try the original PoC (if available) and observe if the unpatched version exhibits the same error as the buggy version at runtime (e.g., kernel panics), unfortunately the original PoC only works in 35% of the cases. To holistically and automatically validate unpatches, we apply a validation pipeline that employs a multi-phase approach integrating reachability analysis, PoC testing, generic fuzzing, and patch-correctness.

At first, we conduct a reachability analysis to determine if the introduced vulnerability can be accessed from the entry point. Using LLVM IR obtained from compilation, we perform control flow analysis to check if the code changes in the patch are reachable. If unreachable, the patch is classified as a failure case; otherwise, we proceed to the next phase.

Next, we apply the revert patch to the target program and execute the original PoC to observe if it triggers any crashes. If crashes occur, we evaluate them against our **Validation Criteria**. If no crashes are detected, we advance to the next verification method. When the original PoC fails to trigger crashes, we employ a customized Syzkaller for further validation. The original PoC or reproducer serves as the initial corpus to accelerate the process. Given the limited scope of our revert patch and its targeting of vulnerability-related locations, we focus on mutating existing syscall sequences rather than generating random mutations. We disable the generation of new programs in Syzkaller and adjust the mutation rate (fixed to 2.0) to ensure all mutations occur on existing syscall sequences, including the addition of new syscalls. Each Syzkaller instance runs for a predetermined duration per patch. Any crashes discovered during this fuzzing process are subsequently subjected to evaluation in the **Validation Criteria** below.

Furthermore, to reduce the risk of unintended side effects (*e.g.,* breaking functionality or introducing new errors), we employ both LTP and KLAUS (the state-of-the-art patch correctness assessment) [44]. KLAUS not only fully utilizes the original PoC and the reverted patch to explore different contexts that trigger the introduced vulnerability, but also helps discover possible contexts that lead to unintended errors. Similarly, any crashes identified by KLAUS are then subjected to evaluation based on the Validation Criteria below.

Validation Criteria. On one hand, for all crashes with PoC(s), we would compare the crash behaviors on the target program and the unpatched program to determine if the crash is caused by the reverted patch. If yes, continue; Otherwise, the validation process is ended. Then, if the reverted patch has no additional hunks compared to the original patch, we can safely conclude that the same vulnerability is introduced into the target program. However, if the reverted patch has some additional hunks, we would manually analyze these additional hunks to identify if the reverted patch brings back the correct vulnerability. On the other hand, if no PoC is generated, we first match the error type and fault location between the new crash and original vulnerability to confirm alignment with the reverted patch; second, we compare the execution paths and operational contexts to ensure consistency in crash-triggering conditions; third, we align the memory error

specifics (object type, access type, address space) with the original vulnerability; finally, we validate the crash originates from the reverted patch to confirm successful reintroduction.

Ensuring triggerability and checking for side-effects. Table 3 reports the proportion of unpatches validated through different stages across two Linux kernel versions. In both versions, the original PoC is valid for only about 35% of all unpatches that we manually confirmed to be correct. Syzkaller-based PoC-directed fuzzing identifies an additional 14% of bug-triggering inputs, whereas Klaus-based fuzzing is comparatively ineffective, contributing only about 1% more.

Through this multi-stage validation process, we ensure that the exposed vulnerability in the unpatched version can be triggered via ❶ the original PoC, ❷ Syzkaller-based fuzzing seeded by the original PoC, or ❸ KLAUS-based fuzzing guided by the patch. Moreover, we further check that the crash-inducing input is indeed triggering the intended vulnerability rather than introducing unrelated bugs. Finally, the validation checks that such inputs do not crash the kernel version preceding the unpatched one, thereby reducing the risk of unintended side effects.

Algorithm 3: Greedy selection for reverted patch merging

Input: Verified reverted patches $\mathcal{P}$; optional batch-size limit K
Output: A conflict-free merge batch $B \subseteq \mathcal{P}$

$I \leftarrow$ empty map from file to set of patches
foreach $p \in \mathcal{P}$ **do**
  foreach $f \in \mathsf{ModifiedFiles}(p)$ **do**
    $I[f] \leftarrow I[f] \cup \{p\}$
foreach $p \in \mathcal{P}$ **do**
  $\mathsf{Conf}(p) \leftarrow \bigcup_{f \in \mathsf{ModifiedFiles}(p)} (I[f] \setminus \{p\})$
$R \leftarrow \mathcal{P}$; $B \leftarrow \emptyset$; $C \leftarrow \emptyset$
while $R \neq \emptyset$ ***and*** *(K is unset* ***or*** $|B| < K$*)* **do**
  foreach $p \in R$ **do**
    $\Delta(p) \leftarrow |\mathsf{Subsystems}(p) \setminus C|$
    $\Gamma(p) \leftarrow |\mathsf{Conf}(p) \cap R|$
  Choose $p^\star \in R$ maximizing $\Delta(p)$, then minimizing $\Gamma(p)$
  $B \leftarrow B \cup \{p^\star\}$
  $C \leftarrow C \cup \mathsf{Subsystems}(p^\star)$
  $R \leftarrow R \setminus (\{p^\star\} \cup (\mathsf{Conf}(p^\star) \cap R))$
return B

5.2 Reverted Patch Merging

Following the validation stage, a substantial collection of reverted patches applicable to the target program is typically obtained. To streamline benchmark testing and enable vulnerability customization, we propose a patch selection strategy that merges multiple vulnerabilities into a single version while resolving patch conflicts. This strategy takes as input the set of verified reverted patches, along with their associated metadata (primarily the list of modified files and derived subsystem information). It employs a greedy algorithm designed to construct a maximal conflict-free subset of patches, ensuring that no two selected patches conflict by modifying the same file, thereby producing a coherent and valid patchset for benchmark creation.

To formally represent these file-level conflicts, we model the relationships between patches using a *conflict graph*, denoted as $G = (V, E)$. In this graph, the set of vertices V corresponds to the individual reverted patches available for selection. An undirected edge $(p_i, p_j) \in E$ exists between two distinct vertices (patches) p_i and p_j if and only if their sets of modified files have a non-empty intersection (i.e., $\mathsf{ModifiedFiles}(p_i) \cap \mathsf{ModifiedFiles}(p_j) \neq \emptyset$). This file-level strategy is conservative: even non-overlapping hunks within the same file are treated as conflicting, which reduces merge failures in practice. Therefore, two patches connected by an edge are mutually exclusive and cannot be selected together; any valid selection must avoid adjacent vertices in G.

The selection of patches represents a Maximal Independent Set problem [27] on the conflict graph, which we resolve using an iterative greedy algorithm, as shown in Algorithm 3. Initially, all verified reverted patches form the candidate set. In each iteration, the algorithm evaluates

the remaining candidate patches based on a predefined scoring function that prioritizes diversity and conflict minimization. Specifically, patches are ranked higher if they introduce coverage of new kernel subsystems not yet represented in the currently selected set. Among patches offering similar diversity benefits, those with fewer conflicts (*i.e.,* a lower degree in the subgraph induced by the remaining candidates) are preferred. If a tie remains, we randomly select one candidate to proceed. The top-ranked patch according to this heuristic is selected and added to the final patchset. Subsequently, this selected patch, along with all other candidate patches directly conflicting with it (its neighbors in the conflict graph), are removed from the candidate pool. This iterative selection and pruning process continues until the candidate pool is exhausted or a predefined termination condition (*e.g.,* reaching a maximum desired number of patches) is met, yielding a final set of mutually non-conflicting patches that encourages subsystem diversity.

5.3 Implementation Highlights

pPatch Partner works collaboratively with pPatch Core §3.5, and is also implemented in Python with 4480 LoC. With the help of pPatch Core, pPatch Partner orchestrates automated patch application, kernel compilation, and vulnerability testing. Initially, it leverages pPatch Core to generate a reversed patch. Subsequently, it automates patch application, incorporating a retry mechanism to mitigate compilation failures. Validation then ensures patch functionality by comparing results against expected outputs, confirming that the modifications align with the intended vulnerability reconstruction. During validation, crash monitoring employs QEMU to execute Proof-of-Concept (PoC) exploits against the patched kernel, systematically detecting crashes and anomalies. Crash logs are recorded, and the database is updated with detailed reports, providing insights into patch effectiveness and potential exploitability. Notably, Partner utilizes **OverlayFS** to significantly accelerate kernel compilation and reduce storage overhead. By applying modifications on a separate writable layer, rather than duplicating the entire source tree, only changed compilation artifacts are stored. This approach minimizes disk usage and avoids redundant recompilation, resulting in faster building and improved resource efficiency.

6 Effectiveness of The pPatch Benchmark

In this section, we evaluate how useful pPatch can help in producing realistic vulnerability benchmarks from two perspectives:

RQ3 How effective is pPatch in reverting **multiple** patches and fuse a large set of vulnerabilities into the same target version? This allows pPatch to generate customizable benchmarks.

RQ4 Can pPatch serve as a practical benchmark for kernel-oriented vulnerability detectors?

6.1 Group Unpatching (RQ3)

To evaluate the practicality of pPatch, we measure its performance in unpatching and consolidating a large number of real-world vulnerabilities into a unified version of the Linux kernel (v6.6.70). This unpatched kernel will serve as a benchmark for kernel-oriented security tools (see §6.2 for details).

Multiple vulnerabilities merged into a unified version. With the greedy algorithm in the conflict-aware Reverted Patch Merging component, pPatch selected verified unpatch candidates from a pool of 187 vulnerabilities for v6.6.70, and successfully integrated a substantial subset of 157 verified vulnerabilities into one coherent kernel image. The resulting consolidated benchmark exhibits significant diversity, crucial for realistically evaluating the breadth of security tools. These 157 vulnerabilities span across 78 distinct kernel subsystems, offering wide coverage of kernel functionalities. Notably, this includes vulnerabilities within 30 specific device driver categories (*e.g.,* covering networking, GPU, HID, media, and Bluetooth) , which are rich sources of historical

bugs. Furthermore, the integrated vulnerabilities represent 7 distinct Common Weakness Enumeration (CWE) categories, ensuring the benchmark encompasses a variety of fundamental flaw types. This outcome highlights pPatch's capability to move beyond single unpatches and automate the creation of large-scale, heterogeneous vulnerability benchmarks that mirror real-world complexity, providing a robust platform for rigorous security tool assessment.

UAF-focused benchmark. To support specialized evaluation (e.g., to evaluate fuzzers or runtime sanity checkers that targeting use-after-free (UAF) bugs), we generate a UAF-only benchmark by filtering vulnerabilities to unpatch on top of the greedy algorithm. This collection includes 78 UAF vulnerabilities merged into dedicated kernel version (v6.6.70), showcasing the potential of using pPatch as a benchmark generator that can generate tailored benchmarks on users' demand.

6.2 Applicability (RQ4)

We evaluate several security tools using the KVulnBench benchmark generated by pPatch. As kernel fuzzing requires extensive computational resources, to complete the experiments with reasonable resources without losing diversity and balance of vulnerability, we curate a representative subset of 100 vulnerabilities from KVulnBench, covering 74 subsystems and 6 CWE categories, thus ensuring comprehensive coverage for general-purpose vulnerability benchmarking. We evaluate the performance of state-of-the-art security tools using two key metrics [17, 34]: ❶ Detection Rate (DR),*i.e.,* which measures the proportion of kernel vulnerabilities in KVulnBench that can be discovered by these tools; ❷ Time-to-Vulnerability (TTV), *i.e.,* which quantifies the time required for kernel fuzzers to discover these vulnerabilities.We summarize all reported DR/TTV in Table 4.

Setup. We consider three families of detectors: coverage-guided fuzzers (Syzkaller-Seeded and Syzkaller-unSeeded modes, and Healer), a representative directed fuzzer (SyzDirect), and static analyzers (Smatch, Coccinelle, and CodeQL). We select Smatch and Coccinelle in accordance with the Linux kernel documentation's recommendations on commonly used static analysis tools for kernel development [7], and omit Sparse because Smatch builds upon Sparse and largely subsumes its checks [5]. In addition, we include CodeQL as a representative query-based static analyzer. Appendix B summarizes the tool versions and configurations used in our evaluation. All experiments were run on the server, equipped with dual 24-core Intel Xeon Gold 6248R CPUs, 512GB of RAM running Ubuntu 24.04 LTS.

❶ Coverage-guided fuzzers were evaluated over 3 rounds of 48-hour fuzzing campaigns, each with 108 VM instances with 2 CPU cores and 2 GB of memory (31,104 CPU-hours in total). To mitigate potential side effects from Syzkaller's test cases, we compare the performance `with` and `without` test cases (*i.e.,* Syzkaller-Seeded and Syzkaller-unSeeded) to provide benchmark testing results. ❷ Directed fuzzers (*i.e.,* SyzDirect) targets 30 vulnerabilities randomly picked from the KVulnBench, and conduct fuzzing with two 48-hour rounds per target, accumulating 5760 CPU-hours overall for this phase. ❸ static analyzers run once on the KVulnBench as the results are deterministic. Crash attribution combines automatic bisection on the reverted-patch set with manual triage for non-reproducible cases (details in Appendix B).

Results and Findings. ❶ Among **coverage-guided fuzzers**, Syzkaller-unSeeded slightly outperformed the seeded version in detection rate (37% vs. 35%), although it achieved lower coverage. This indicates that coverage alone is not a reliable predictor for vulnerability rediscovery in this benchmark. Healer, while converging much faster (median TTV <6h), exhibited a lower detection rate, highlighting the trade-off between speed and depth in vulnerability detection. For ❷ **directed fuzzers** SyzDirect successfully detected 14 out of these 30 targeted vulnerabilities (46.7%) in at least one of the 48-hour session. Additionally, during the directed fuzzing process, SyzDirect uncovered 15 vulnerabilities from the remaining 70 non-targeted cases, further demonstrating the robustness

Table 4. **Cross-detector results on KVulnBench (Linux v6.6.70).**

Detector	Type	DR	TTV (h)	Coverage
Syzkaller-Seeded	Coverage-guided fuzzer	35/100	18.6	300,639
Syzkaller-unSeeded	Coverage-guided fuzzer	37/100	18.2	255,197
Healer	Coverage-guided fuzzer	12/100	< 6	249,263
SyzDirect	Directed fuzzer	14/30	—	—
Smatch	Static analyzer	1/100	—	—
Coccinelle	Static analyzer	0/100	—	—
CodeQL (standard & kernel-spec)	Static analyzer	9/100	—	—

of the benchmark and validating the correctness of the pPatch. ❸ Among **static analyzers**, Smatch detected one vulnerability, while Coccinelle and the default CodeQL queries (ql) found none. A kernel-specialized CodeQL query [10] collection detects 9 vulnerabilities.

In summary, the results reveal distinct strengths across the different detectors. Coverage-guided fuzzers excel in broad vulnerability discovery over extended periods, while directed fuzzers like SyzDirect are highly effective in quickly identifying vulnerabilities within the targeted set. Static analyzers, particularly those with specialized queries, provide a complementary approach, but their effectiveness in general vulnerability detection remains limited within this benchmark.

7 Discussion and future work

Versatility. Beyond vulnerability detection, our benchmark serves to evaluate a wide range of software security tools, *e.g.,* APR tools [45], hot-patch solution [46], symbolic execution [26], new security mitigations [6], vulnerability sanitizers [15]. And our customized benchmark could help evaluate one specific subsystem (*e.g.,* USB) or CWE type (*e.g.,* UAF). All vulnerabilities in our benchmark, by nature, have their security patch - the reversion of the reverted patch. APR tools can take historic vulnerabilities and our reverted patches as ground truth to evaluate effectiveness. While this work emphasizes vulnerability detection, we defer exploration of these additional security applications to future research.

Benchmark Validation Methodology. Current evaluations of kernel fuzzers rely extensively on manual analysis when tools fail to generate actionable crash reproducers. While MAGMA manually constructs trigger conditions to verify vulnerabilities, this approach faces unique challenges in the kernel context: unlike user-space programs, reproducing precise execution paths to trigger kernel vulnerabilities is inherently more complex, requiring labor-intensive and time-consuming manual effort. To address this gap, future work could integrate eBPF to trace execution states and synthesize trigger conditions, enabling systematic validation of vulnerability triggerability [43].

Semantic Patch. Currently, pPatch operates at the granularity of hunks and employs a conservative selection strategy. However, hunks are defined by code change locations rather than semantic coherence, which means a single hunk may inadvertently include unrelated code fragments and introduce unintended side effects during unpatching. In future work, we plan to integrate semantically-aware, fine-grained code differencing tools such as GumTree [9] and ClDiff [14].

8 Related Work

Fuzzing Evaluation. Evaluating the effectiveness of fuzzing tools is a non-trivial task, and researchers have developed a range of methodologies to assess and compare different fuzzers.

Some existing methods typically rely on a single metric, such as code coverage [2, 33, 36], crash counts [13, 16, 28], or mutant detection [11]. However, each metric has limitations: coverage may not reflect actual bug-finding [3, 17], bug counts depend on deduplication and target selection, and mutants may lack realism.

Recent platforms like FuzzBench [28] and UNIFUZZ [18] offer standardized benchmarks and multi-metric evaluation. FuzzBench focuses on coverage and statistical analysis over real-world targets, while UNIFUZZ integrates 35 fuzzers and six metric types, emphasizing that no single fuzzer or metric suffices. The SoK study [34] further highlights best practices (*e.g.,* long runs, real bugs, statistical tests) and cautions against synthetic corpora like LAVA-M. Our work complements these by proposing pPatch, a benchmark built from historical vulnerabilities via patch reversing. pPatch offers concrete, ground-truth targets, enabling more realistic and focused evaluation.

Vulnerability Injection. Vulnerability injection aims to create synthetic yet realistic bugs in software for benchmarking, training, and evaluating security tools. Unlike natural bugs that are hard to collect at scale with ground-truth labels, injected vulnerabilities offer controllable and repeatable test conditions.

Early work like LAVA [8] pioneered automatic vulnerability injection by planting memory bugs (e.g., buffer overflows) in real programs using contrived "trigger" conditions. It enabled controlled evaluations with known bug locations but was later criticized for its narrow bug types and unrealistic injection patterns that may not generalize to real-world flaws. More recent efforts, such as VULGEN [30], propose a hybrid approach that mines vulnerability-inducing patterns from real-world patches and uses a neural model to predict likely injection points. By decoupling "what to inject" from "where to inject", it enables diverse CWE-class bugs (18+) with a 69% success rate, generating over 686,000 samples to enhance ML-based detector training. Built on VULGEN's foundation, VinJ [31] scales up vulnerability injection by automating the entire process. It demonstrates the feasibility of combining learned injection patterns with large-scale automation, bridging research on code transformation, machine learning, and software security. Unlike prior vul-injection approaches, pPatch injects vulnerabilities by reverting real-world patches, thereby preserving the authenticity of injected bugs. This enables more realistic and reliable evaluation, benchmarking, and training for vulnerability analysis tools.

9 Conclusion

The complexity of unpatching historical vulnerabilities into modern codebases hinders realistic benchmarking for security tools. We developed pPatch, an automated framework to systematically revert patches while aiming for minimal unintended semantic disruption. pPatch proved effective, validating 187 historical vulnerabilities in recent Linux kernels (significantly outperforming baselines) and replicating over half of MAGMA's manual user-space unpatches. This success enabled the creation of KVulnBench, a novel benchmark resource derived from these verified vulnerabilities.

Analysis using KVulnBench showed dynamic techniques significantly outperform static analysis, with directed fuzzing achieving approximately 47% success on targeted bugs. Limitations remain, primarily concerning complex code evolution scenarios and comprehensive validation guarantees against side-effects. Future work will focus on enhancing pPatch's robustness and broadening its applicability. Overall, pPatch provides a valuable automated methodology for generating realistic benchmarks like KVulnBench, enabling more rigorous security tool assessment and fostering software security advancements.

10 Data Availability

pPatch and pPatch-Partner is open-sourced at https://github.com/OS3Lab/pPatch.

References

[1] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*.

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security (CCS '17)*.

[3] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*. 1621–1633.

[4] Matej Bölcskei, Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2025. Encarsia: Evaluating cpu fuzzers via automatic bug injection. In *Proceedings of the 34st USENIX Security Symposium (USENIX Security 25)*.

[5] Dan Carpenter. [n. d.]. Smatch. https://smatch.sourceforge.net/.

[6] Kees Cook. 2024. Per-call-site slab caches for heap-spraying protection. https://lwn.net/Articles/986174/.

[7] Linux Kernel Documentation. [n. d.]. Linux Kernel Testing. https://docs.kernel.org/dev-tools/testing-overview.html.

[8] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. In *2016 IEEE symposium on security and privacy (SP)*.

[9] Jean-Remy Falleri and Matias Martinez. 2024. Fine-grained, accurate and scalable source differencing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*.

[10] Fullway Wang. 2025. QlRules: Linux Rules Directory. https://github.com/fullwaywang/QlRules/tree/main/linux.

[11] Philipp Görz, Björn Mathis, Keno Hassler, Emre Güler, Thorsten Holz, Andreas Zeller, and Rahul Gopinath. 2023. Systematic assessment of fuzzers using mutation analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4535–4552.

[12] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. MAGMA: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (Dec. 2020), 29 pages. doi:10.1145/3428334

[13] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 36–50.

[14] Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, and Wenyun Zhao. 2018. ClDiff: generating concise linked code differences. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*.

[15] Linux Kernel. 2020. Kernel Concurrency Sanitizer (KCSAN). https://docs.kernel.org/dev-tools/kcsan.html.

[16] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings of The Network and Distributed System Security Symposium (NDSS '20)*.

[17] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*.

[18] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security '21)*.

[19] Linux Kernel. 2018. tracing: Check for no filter when processing event filters. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=70303420b5721c38998cf987e6b7d30cc62d4ff1.

[20] Linux Kernel. 2019. qmi_wwan: Fix out-of-bounds read. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=904d88d743b0c94092c5117955eab695df8109e8.

[21] Linux Kernel. 2020. qmi_wwan: unconditionally reject 2 ep interfaces. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=00516d13d4cfa56ce39da144db2dbf08b09b9357.

[22] Linux Kernel. 2020. tcp: tcp_v4_err() icmp skb is named icmp_skb. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=239174945dac8cb9613db7755103d5fb6c32241d.

[23] Linux Kernel. 2022. net: rds: acquire refcount on TCP sockets. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3a58f13a881ed351198ffab4cf9953cf19d2ab3a.

[24] Linux Kernel. 2025. af_unix: fix use-after-free in unix_stream_read_actor(). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4b7b492615cf3017190f55444f7016812b66611d.

[25] Linux Kernel. 2025. Linux Kernel CVE information. https://git.kernel.org/pub/scm/linux/security/vulns.git.

[26] Jian Liu, Lin Yi, Weiteng Chen, Chengyu Song, Zhiyun Qian, and Qiuping Yi. 2022. LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security 22) (USENIX Security '22)*.

[27] Michael Luby. 1985. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*. 1–10.

[28] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering*

conference and symposium on the foundations of software engineering. 1393–1403.

[29] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *Proceedings of the 27th USENIX Conference on Security Symposium (USENIX Security '18)*.

[30] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2023. Vulgen: Realistic vulnerability generation via pattern mining and deep learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2527–2539.

[31] Yu Nong, Haoran Yang, Feng Chen, and Haipeng Cai. 2024. VinJ: An Automated Tool for Large-Scale Software Vulnerability Data Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 567–571.

[32] Linux Test Project. [n. d.]. Linux Test Project. https://github.com/linux-test-project/ltp.

[33] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS '17)*.

[34] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. Sok: Prudent evaluation practices for fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1974–1993.

[35] Ridwan Shariffdeen, Xiang Gao, Gregory J Duck, Shin Hwei Tan, Julia Lawall, and Abhik Roychoudhury. 2021. Automated patch backporting in Linux (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21)*.

[36] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS '17)*.

[37] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*.

[38] Syzbot. 2016. Syzkaller Dashborad. https://syzkaller.appspot.com/.

[39] Syzbot. 2020. KASAN: null-ptr-deref Write in event_handler. https://syzkaller.appspot.com/bug?id=28cccdd18b4bb8670d077937fb8d4849dca96230.

[40] Syzkaller. 2019. KASAN: global-out-of-bounds Read in qmi_wwan_probe. https://syzkaller.appspot.com/bug?extid=b68605d7fadd21510de1.

[41] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. 2023. SyzDirect: Directed Greybox Fuzzing for Linux Kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*.

[42] Dmitry Vyukov. 2016. Syzkaller. https://github.com/google/syzkaller.

[43] Zicheng Wang, Yueqi Chen, and Qingkai Zeng. 2023. PET: prevent discovered errors from being triggered in the linux kernel. In *Proceedings of the 32nd USENIX Conference on Security Symposium (USENIX Security '23)*.

[44] Yuhang Wu, Zhenpeng Lin, Yueqi Chen, Dang K Le, Dongliang Mu, and Xinyu Xing. 2023. Mitigating Security Risks in Linux with KLAUS: A Method for Evaluating Patch Correctness. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security '23)*.

[45] Yunlong Xing, Shu Wang, Shiyu Sun, Xu He, Kun Sun, and Qi Li. 2024. What IF is not enough? fixing null pointer dereference with contextual check. In *Proceedings of the 33rd USENIX Conference on Security Symposium (USENIX Security '24)*.

[46] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. 2020. Automatic hot patch generation for android kernels. In *Proceedings of the 29th USENIX Conference on Security Symposium (USENIX Security '20)*.

[47] Su Yang, Yang Xiao, Zhengzi Xu, Chengyi Sun, Chen Ji, and Yuqing Zhang. 2023. Enhancing OSS Patch Backporting with Semantics. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*.

[48] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. 2022. FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security '22)*.

A Failure Analysis for RQ1

Analysis of failed cases. pPatch only successfully unpatches 395 (64%) out of all 614 cases we have collected, and the success rate is even lower 187 (30%) if we consider the auto-generation of bug-triggering input as necessary. We examined these failed cases and summarized the failure reasons as follows.

1) Context not found. Files and directory structures in the Linux kernel are routinely deleted, merged, or renamed, and this causes pPatch to completely miss cases where the original patch applies to a file that no longer exists in the target version. For example, code related to the `io_uring` module has been moved to a new directory, and, pPatch is unable to determine the current location of any patch that applies to the old `io_uring` directory, making it impossible to even start the unpatching process.

2) Symbol lookup failures caused by macro expansion. When using macros to define code snippets, the expanded code may introduce variable names that differ from those in the source code, resulting in discrepancies between the variable names shown in compiler error messages and those in the original source. In such cases, pPatch fails to correctly locate the hunk containing the missing symbol. Take commit@703034 [19] for instance: the compiler reports missing definitions for the variables `FILT_ERR_ERRNO` and `FILT_ERR_NO_FUNCTION`. In `kernel/trace/trace_events_filter.c`, the macro list ERRORS defines error types that are expanded using another macro C, which adds the `FILT_ERR_` prefix during expansion. As a result, the original symbol names are actually `ERRNO` and `NO_FUNCTION`. Since pPatch is currently unable to recognize such macro-induced transformations during symbol lookup, it fails to select the necessary hunk for unpatching.

3) Excessive hunk reversal. A common reason that causes non-triggerable unpatches or side-effects are excessive hunk reversal, i.e., when the granularity of a hunk, as interpreted by pPatch during the reversal process, is excessively large, encompassing lines of code unrelated to the specific fix.

This is observed with commit@3a58f1 [23], which patches `net/rds/tcp.c` by adding logic to properly manage net namespace reference counts. In this case, the reversal mechanism identified a hunk containing the newly added reference counting block. However, due to the way the patch context was defined or interpreted, the reversal operation also removed an adjacent, pre-existing line: `rtn = net_generic(net, rds_tcp_netid);`. This line is crucial as it initializes the `rtn` variable used in subsequent conditional checks and assignments (`if (rtn->sndbuf_size > 0)...`). By erroneously removing this essential initialization, `pPatch` generated code where `rtn` remains uninitialized or holds a stale value within that scope. This fundamentally alters the control flow and data state, corrupting the logic surrounding the original vulnerability and rendering the generated code incapable of reproducing the targeted security flaw. This failure underscores the difficulty in precisely delineating the boundaries of a fix within its surrounding code context during automated reversal.

4) Incomplete hunk reversal. When GNUPatch does not reverse all hunks of a complex patch due to its conservative nature (see §3.2 for details), it can lead to compilable but semantically incorrect code, especially when there are concurrency primitives involved, and again, leading to non-triggerable unpatches or side-effects,

An example was observed when attempting to reverse commit@4b7b49 [24], which addressed a use-after-free vulnerability in `net/unix/af_unix.c`. The original patch modified locking behavior across multiple hunks of the code. During reversal, pPatch successfully reintroduced a `spin_lock(&sk->sk_receive_queue.lock)` call, but failed to reintroduce

the matching `spin_ unlock(&sk->sk_receive_queue.lock)` in a specific execution path. This omission stems from the conservative nature in hunk selection in pPatch, which fails to cherry-pick

the unlock operation in the commit. The resulting generated code suffers from a lock imbalance, and this error introduces a new flaw (potential deadlocks)—a side-effect.

5) Vulnerabilities fixed by multiple commits pPatch inherently assumes a one-to-one mapping between a reported vulnerability and a single fixing commit. However, in practice, complex vulnerabilities or those discovered and fixed incrementally may require multiple commits to fully address the underlying issue(s). A concrete example is commit@28cccd [39] which are fixed by three separate commits. pPatch, when provided with only one of these commit identifiers, will only reverse the changes associated with that specific commit. This partial reversal is insufficient to expose the original vulnerability, making it untriggerable.

6) Erroneously labeled fix commit While uncommon, the fix commit for a vulnerability can be mistakenly labeled on syzbot. This leads pPatch to generate a completely irrelevant unpatch. Even worse, when the fix commit is mistakenly pointed to a buggy commit, as in the case of commit@239174 [22], pPatch can generate a fix, which, when applied to the new version, has no effect.

Analysis of failed cases of Backporting Tools. FixMorph and TSBPORT perform poorly in the forward-porting setting. In what follows, we analyze the root causes of their failures with concrete cases, and illustrate how pPatch overcomes some of these issues.

1) Format Issues. FixMorph and TSBPORT may generate patches with non-standard formatting, causing standard GNUPatch utilities to fail to parse or apply them. For example, when reverting patch@1997b3 on Linux kernel v6.12.16, TSBPORT produces a diff fragment using `104,105d` to indicate the deletion of lines 104–105 in `net/dns_resolver/dns_key.c`. However, it does not provide the starting line number in the target version after deletion, and it only includes one line of content rather than the two lines that should be deleted. As a result, the patch cannot be parsed correctly and the reversion fails. pPatch emits patches in the standard unified diff format. In each hunk header, the affected line ranges are explicitly determined by the starting line number and the number of added/deleted lines, following the unified-diff specification; thus pPatch does not suffer from patch-formatting issues.

2) Injected Extra Headers. During reversion, TSBPORT may inject additional header inclusions. Such headers may be absent in the target version or may no longer contain the required definitions, leading to compilation failures. When reverting patch@18f547 on Linux kernel v6.12.16, TSBPORT replaces `memcpy_and_pad` with `memcpy`. In the version where patch@18f547 was introduced, `memcpy` is declared in `include/asm/unaligned.h`, whereas in the current version it is provided by `include/linux/unaligned.h`. TSBPORT fails to recognize this evolution and consequently injects `include/asm/unaligned.h` during reversion. Since this header does not exist in the current version, the build fails. pPatch only adds code by selecting hunks from the commit history and follows a conservative selection policy: a hunk is selected only if it overlaps with the patch being reverted, or if it modifies a symbol that is involved in the reverted patch. Therefore, pPatch does not introduce unnecessary header files in this case.

3) Syntactically Incomplete Patches. Both TSBPORT and FixMorph may generate patches that are syntactically incomplete. For instance, when processing patch@eb7319, the patch produced by TSBPORT contains a deleted line ending with `)`, but the corresponding added line does not end with `)`, leaving parentheses unbalanced and causing syntax errors at compilation time. pPatch adds code at the granularity of hunks, and each selected hunk corresponds to a self-contained code block, which helps preserve syntactic well-formedness.

4) Usage of Outdated Variables. FixMorph may generate patches that reference variables that no longer exist or whose definitions have significantly evolved. When reverting patch@0ddc51, the patch generated by FixMorph references a non-existent variable `neted`. We cannot find a

definition of `neted` in the current codebase or in the relevant history, which leads to compilation failures. In contrast, hunks added by pPatch strictly originate from the repository history, and thus it does not introduce references to non-existent or outdated variables. In this case, pPatch selects two conflict hunks from commit@c56f9e and commit@dca54a to remove code blocks that were newly introduced in the current version compared to the version where patch@0ddc51 was applied, thereby successfully reverting the changes in patch@0ddc51.

B Additional Details for RQ3

Crash Attribution and Validation. We attribute crashes to intended vulnerabilities via a two-stage procedure. For reproducible cases (C/syz reproducers), we perform automatic bisection over the set of reverted patches: starting from the full set, we iteratively apply halves of the set to the kernel under test and run the reproducer, converging to the single causal reverted patch in $O(\log n)$ steps. For non-reproducible cases, we manually inspect call traces and kernel logs, cross-referencing the touched subsystems, functions, and change hunks with the ground-truth metadata for the 100 vulnerabilities. A small number of ambiguous cases are conservatively discarded.

Directed Fuzzing Setup. SyzDirect requires specifying targets through source-level instrumentation near the vulnerability execution path. For each of the 30 randomly selected targets, we instrument the kernel and run two independent 48-hour campaigns (96 hours per target). SyzDirect confirms 14/30 targets and additionally triggers 15 distinct non-target vulnerabilities present in the instrumented build. Unless otherwise stated, we report the targeted confirmations as the primary metric; the incidental non-target discoveries are recorded but not counted toward target DR.

Static Analyzer Configuration. We follow the selection rationale in Section 6.2 and report tool versions and invocations here. Smatch is built from commit `34981e` and run with default kernel integration. Coccinelle is executed via `make coccinelle` in the kernel build system. CodeQL uses the official CLI (v2.20.7). We run (i) the standard C/C++ query suite (`cpp/ql`), which yields no detections on our set, and (ii) a kernel-specialized query collection (~5.2k queries) tailored to historical CVEs, which yields 9 detections. Because the latter encodes vulnerability-specific patterns, these results should be interpreted as upper bounds under specialization and not as evidence of generalizable static efficacy.

Coverage Measurements. We record basic coverage counters for completeness but do not treat coverage as a primary endpoint. In our runs, Syzkaller-Seeded achieves the highest coverage yet a slightly lower DR than Syzkaller-unSeeded, reinforcing that gross coverage is not strongly correlated with bug rediscovery on this benchmark.

Received 2026-02-25; accepted 2026-03-24