Not In My Git Yard: Catching Backdoors at Commit and Release Time

Dimitri Kokkonis✉
dimitri.kokkonis@cea.fr
Université Paris-Saclay
CEA, List
Paris-Saclay, France

Michaël Marcozzi
michael.marcozzi@cea.fr
Université Paris-Saclay
CEA, List
Paris-Saclay, France

Stefano Zacchiroli
stefano.zacchiroli@telecom-paris.fr
LTCI, Télécom Paris
Institut Polytechnique de Paris
Palaiseau, France

Abstract

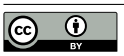
Code-level backdoors—stealthy code changes that grant hidden privileges via secret triggers—pose a persistent threat to open-source software. Known attempts to inject such backdoors into widely used projects through malicious commits, tampered release packages, or compromised third-party dependencies, were stopped only by luck and manual review. Existing Continuous Integration (CI) pipelines cannot detect these attacks, and downstream binary analysis tools require substantial manual effort. In this work, we present Lily, an automated approach that strengthens open-source development and release processes against backdoor injection. Lily integrates a backdoor detection mechanism into (1) CI pipelines to block malicious commits, and (2) release vetting workflows to prevent tampered releases or compromised dependencies from entering large ecosystems, such as Linux distributions. Lily offers two key contributions. First, it enhances CI-compatible fuzzing with the capability to detect triggers of suspicious behavior based on historical and current software executions. This enables fast, precise backdoor detection suitable for both CI and update validation workflows. Second, it combines code change analysis with fuzzing data to precisely point maintainers to backdoor-revealing code regions, even when release updates modify millions of lines of code. We also outline five strategies attackers could use to evade Lily, and evaluate corresponding defenses. Our experiments across hundreds of benign and backdoored commits and releases show that Lily achieves high detection accuracy with low false alarm rates, reliably identifies malicious code, resists adversarial attempts, and would have prevented real-world backdoor incidents.

CCS Concepts

• **Security and privacy → Software security engineering**; *Malware and its mitigation*; • **Software and its engineering → Software testing and debugging**.

Keywords

Fuzzing, Dynamic Analysis, Backdoors, Continuous Integration, Release Vetting, Software Supply Chain Security



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834352

ACM Reference Format:
Dimitri Kokkonis, Michaël Marcozzi, and Stefano Zacchiroli. 2026. Not In My Git Yard: Catching Backdoors at Commit and Release Time. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834352

1 Introduction

Context. *Code-level backdoors* [21, 45]—hidden functionalities embedded within source code that allow individuals aware of their presence to obtain elevated privileges or unauthorized features via secret triggers—remain a persistent threat to open-source software. Incidents involving malicious commits to the PHP public repository [1], tampering with ProFTPD and vsFTPd release packages [28, 29], or the injection of backdoors through a compromised dependency to XZ Utils [30] have gone undetected for days, being uncovered only through a combination of manual inspection and chance. The consequences of missing such threats are severe, especially as the ever-expanding ecosystem of software dependencies magnifies the potential impact of a single compromise [7].

Problem. The current open-source software development and release stack lacks systematic defenses against backdoor injection attacks. Upstream, *Continuous Integration (CI) pipelines* run automated tests—typically on every commit—to ensure code quality through compilation checks and regression testing. *Fuzzing*, already integrated into CI pipelines of major projects [31, 40–42], targets crashes and memory safety bugs through brute-force testing of the revised code. Because CI must provide rapid feedback, fuzzing runs are brief, often limited to about 10 minutes [10, 17, 20]. Existing CI mechanisms, however, cannot detect backdoored commits such as hard-coded credentials or hidden reverse shells. A recent machine-learning approach that analyzes Git metadata [12] could offer a partial solution but still yields many false alarms. Downstream, state-of-the-art backdoor detection tools [21, 34, 37, 44, 46] help end users vet binaries, but demand manual effort to reverse-engineer code or to dismiss false alarms, limiting scalability.

Goal and Challenges. We aim to harden open-source development and release pipelines by introducing a mechanism that blocks most backdoor injection attacks, integrating it in two stages: (1) in CI pipelines, and (2) in release vetting before updated packages enter major ecosystems such as Linux distributions. Doing so requires addressing three challenges:

(1) CI and release vetting pipelines operate under strict time constraints, so backdoor detection must be fully automated and able to achieve a high detection rate within these limits to warrant

integration. At the same time, any false alarm can halt the development or release process, making high precision essential to maintain developer productivity and preserve maintainer trust.
(2) The code changes to inspect (commit or release diffs) can span thousands to millions of lines, requiring precise reports that pinpoint suspicious locations for CI operators or distribution maintainers.
(3) Broad adoption may prompt adversaries to develop evasion techniques, which must be evaluated and countered.

Proposal. We propose Lily, a new approach for preventing backdoor injection in open-source development. Lily augments existing 10-minute CI fuzzing campaigns with a runtime monitoring mechanism for detecting backdoor triggers, and extends these campaigns to also vet new package releases.

(1) While the revised code is undergoing fuzz testing, Lily monitors its runtime behavior. It flags a potential backdoor only when both of the following conditions are satisfied: (i) it observes a behavior that was absent from the pre-revision code, and (ii) the observed behavior deviates significantly from the revised code's typical behavior, as characterized by executing the revised code on a regression test suite derived from the fuzzing campaigns conducted when vetting prior revisions.
(2) Lily correlates static code change information from commits and releases with dynamic data from fuzzing and monitoring, enabling precise localization of backdoor-revealing code regions.
(3) We systematically evaluate five possible evasion strategies—CI bypass, fuzzer obstruction, segmented or adversarial backdoor patterns and regression test suite poisoning—and assess their feasibility and countermeasures.

Evaluation. We evaluate Lily through 20 ten-minute detection trials on 432 safe and 13 backdoored commits, and on 50 safe and 50 backdoored releases from 13 popular open-source projects. The backdoors come from the Rosarum corpus—real and synthetic—introduced at ICSE'25 as a benchmark for detection tools, following established fuzzing evaluation practices [21, 33]. Lily achieves an average detection rate of 90% at the commit level and 83% at the release level, indicating that it likely would have prevented three real backdoor injection incidents (CVE-2010-20103, CVE-2011-2523, and the PHP attack [1]). False alarm rates remain low—0.2% for safe commits and 4.3% for safe releases on average—and they occur primarily in predictable scenarios such as large merge commits or test harness modifications. Our ablation study confirms that reporting a behavior as a potential backdoor only when it is both novel and atypical dramatically reduces false alarms. Even in worst-case scenarios involving releases differing by thousands to millions of lines, Lily consistently reduced manual review to under a dozen lines, always pinpointing backdoor-revealing code. Our adversarial analysis identifies possible evasion strategies, but they require substantial effort, offer no guaranteed success, and increase attacker exposure. Countermeasures, including our hardened mode, improve resilience and prevent regression test suite poisoning.

Contributions. To sum up, our main contributions are:

(1) Lily, an automated backdoor detection approach which systematically hardens open-source development and release pipelines against injection attacks. During revised code fuzzing, Lily tracks runtime behavior and flags potential backdoors whenever new execution patterns emerge (with regards to the original code) and fall outside the revised code's normal behavior. It further combines code change analysis with fuzzing data to precisely guide maintainers to backdoor-revealing code regions, even in releases modifying millions of lines of code.
(2) A comprehensive evaluation over hundreds of safe and backdoored commits and releases, showing high detection rates with low false alarms, accurate localization of backdoor-revealing code, and robustness under adversarial conditions, including several different attack vectors and countermeasures to them. It also demonstrates that Lily would have prevented several real-world backdoor incidents [1, 28, 29].

2 Background

2.1 Release Cycle and Continuous Integration

In modern software engineering, project development is commonly organized into *release cycles*, during which *releases* (i.e., new versions of a program) are periodically delivered to end users. Each release provides a set of updates (e.g., new features, bug fixes). Between releases, it is standard practice to rely on a Version Control System (VCS) such as Git [14] (whose history is typically public in the case of open-source projects) to facilitate collaboration and ensure traceability. This workflow is frequently complemented by a *Continuous Integration* (CI) system, often called a "CI pipeline." Its primary goal is to provide continuous quality assurance, by automatically executing a sequence of *jobs* (i.e., tasks) which check whether recent changes *committed* to the VCS maintain the integrity of the software. Typical examples include compiling the software and running an associated regression test suite. When a contributor introduces a defect detectable by CI jobs, the CI pipeline reports an error, and typically blocks the integration of the faulty change into the upstream codebase. The individuals responsible for the change are notified, and are expected to inspect the CI logs, diagnose the issue, and submit a correction.

2.2 Fuzzing in Open-Source Development

Graybox fuzzing [15] is an effective automated testing technique that generates test inputs for the Program Under Test (PUT) and uses coverage feedback to guide program exploration [9]. Fuzzers monitor the execution of the PUT—often using sanitizers [38]—and report crashing inputs that reveal issues such as memory errors or assertion failures. Google's OSS-Fuzz [2], launched in 2016, provides free fuzz testing for open source projects, supporting engines such as the AFL++ fuzzer [9]. Developers supply a *fuzzing harness* linking the fuzzer to the project API; for example, libpng's harness [43] loads and processes fuzz-generated PNG files. Harness design strongly affects fuzzing quality, as it determines which parts of the codebase are exercised and under what conditions. As of August 2023, OSS-Fuzz had helped fix over 10,000 vulnerabilities in 1,000 projects. To catch issues as early as possible, some projects also integrate fuzzing in CI pipelines. OSS-Fuzz offers CIFuzz,[1] which builds each commit, runs a brief fuzzing campaign (typically

[1] https://google.github.io/oss-fuzz/getting-started/continuous-integration

10 minutes [10, 17, 20]), and records any crashes with their stack traces and inputs.

2.3 Injecting Code-Level Backdoors

2.3.1 Code-Level Backdoors. While backdoors of many forms have been identified in various components of computer systems [6, 8, 23, 27], this work concentrates on code-level backdoors [21, 45]. These are hidden functionalities embedded directly within the source code of conventional programs, enabling individuals aware of their existence to gain elevated privileges or unauthorized access by triggering them with a secret input. A notable example is the official vsFTPd distribution, which was once reported [29] to include a backdoor. In this case, using the string `":)"` as the FTP username would open a remote root shell on the compromised system hosting the backdoored vsFTPd.

2.3.2 Injecting Backdoors in Open-Source Projects. Although relatively uncommon, code-level backdoor injections in open-source projects have been reported consistently over the years, indicating persistent and organized attempts to compromise software that often underpins critical infrastructure. Among the most notable incidents, attackers infiltrated the XZ Utils library in 2024 [26] (CVE-2024-3094), embedding a multi-stage backdoor into its Git repository. This compromise weakened OpenSSH authentication (as OpenSSH depends on XZ Utils), enabling unauthorized SSH access. The attack was ultimately detected due to performance anomalies. In 2021, the PHP Git server was breached [1], and malicious commits introduced a backdoor into the HTTP server bundled with the PHP interpreter. This vulnerability allowed remote code execution via crafted HTTP headers, but it was detected before reaching an official release, so no CVE was issued. Earlier, between 2010 and 2011, attackers compromised the vsFTPd and ProFTPD distribution servers [28, 29] (CVE-2011-2523 and CVE-2010-20103), distributing releases with hard-coded backdoors for several days.

From such incidents, three primary backdoor injection attack scenarios on open-source projects emerge:

Injection via malicious commits: achieved by getting harmful commits accepted into the upstream repository, or by pushing them from compromised maintainer accounts. This leaves traces in version control history and immediately compromises the software, as seen in the PHP incident.

Out-of-repository injection: achieved by infecting official release packages with the backdoor, as in the vsFTPd and ProFTPD cases.

Software supply-chain injection: a backdoor inserted into a compromised dependency propagates to downstream components, exploiting the complexity of modern dependency networks. The XZ Utils incident illustrates this: embedded in thousands of projects, its backdoor activated only when used by OpenSSH.

3 The Lily Approach

3.1 General Overview

We introduce **Lily**, a novel approach to preventing code-level backdoor injections in open-source projects. At its core, Lily employs a code analysis based on graybox fuzzing which compares two versions of the same PUT. By examining the differences introduced in the newer version, Lily determines whether these changes may include a backdoor.

Lily can be deployed at different stages of open-source development workflows (see discussion in Section 3.2). It consists of three components implementing a three-step approach, which are summarized below and detailed in Sections 3.3 to 3.5. **Section 3.6 illustrates the approach on a concrete example**.

1. **Standard behavior identifier:** establishes a baseline of normal behaviors for the *new version* of the PUT by executing it on a regression test suite derived from fuzzing campaigns for earlier PUT revisions. Inspired by multi-version execution [16] and Rosa [21] (see Section 6), Lily characterizes behavior using system-call profiles, i.e., the sets of system-call types observed during each test execution.[2]
2. **Backdoor detection oracle:** during a fuzzing campaign on the new version of the PUT, this oracle reports a backdoor only if both conditions hold: (i) a fuzzer-generated input triggers a system call profile that differs from the normal profiles identified by the standard behavior identifier, and (ii) the same input does not trigger the corresponding behavior when executed on the previous version of the PUT. Requiring both conditions serves to reduce false alarms by filtering out legitimate new behaviors introduced by the revision (condition 1) as well as atypical behaviors already present in the previous version (condition 2).
3. **Suspicious code tracer:** when the oracle flags a potential backdoor, the tracer highlights the modified source lines that are most likely associated with the suspicious behavior, by filtering out benign changes to produce cleaner, more focused logs. It correlates the non-standard behaviors observed in the new version of the PUT with the source-code differences between the two versions.

3.2 Deploying Lily During Development

Lily can be deployed at either of two (or both) critical stages of the open-source lifecycle to mitigate the three previously described injection scenarios (Section 2.3.2):

During development: Lily is run at the CI stage, complementing existing fuzzing jobs (as in OSS-Fuzz's [2] CIFuzz). By vetting the code changes introduced by each new commit, it provides the capability to block backdoor-injecting commits.

After release updates: Lily is run when end users integrate new versions of an upstream project, to detect supply-chain and out-of-repository injections. *In our evaluation (Section 5), we consider the scenario in which maintainers of major Linux distributions (Debian and Ubuntu) automatically vet new package releases for inclusion in an upcoming distribution version, comparing them against the old package versions they previously shipped (including the same older dependency versions). Unlike vetting individual commits, reviewing a package release means assessing many bundled changes at once. Still, because distribution maintainers must handle large numbers of external packages, similar CI-level constraints apply: short fuzzing time, no manual setup, low false alarms.*

[2] Because system calls mediate all external interactions, any meaningful code-level backdoor must invoke them and can therefore be detected through monitoring (see Sections 4.3 and 4.4 for potential evasions).

Lily is designed to prevent backdoor injections as early as possible in active projects. As such, **when deployed on an existing open-source project, it will only detect backdoors introduced in future commits or releases, not those already present in the codebase**. Detecting preexisting backdoors typically requires a one-time, clean-slate audit involving more extensive analysis, for which state-of-the-art tools such as Rosa [21] are better suited. In practice, a project can first be audited with Rosa and then protected by Lily against future backdoor injections.

3.3 Identifying Standard Behaviors

Graybox fuzzers generate test inputs through mutation: new inputs are produced by randomly modifying existing ones. A feedback loop guides this process. As mutated inputs execute on the PUT, only those that increase control-flow edge coverage are retained as seeds. These seeds then serve as the basis for further mutations, enabling the fuzzer to progressively explore deeper behaviors of the PUT. A campaign begins with a user-provided seed corpus, proceeds through many iterations of mutation and execution, and ultimately outputs the retained seeds.

Integrating Lily into an existing development or release pipeline begins with running an initial graybox fuzzing campaign on the latest commit or release. The resulting seeds are saved as the first regression suite to be consumed by the standard behavior identifier. Starting from this baseline, Lily fuzzes each subsequent commit or release as it becomes available to detect potential injection attempts. For every new version, it reuses all seeds retained during the previous campaign—both as the initial seed corpus for fuzzing and as the regression suite for the standard behavior identifier. This allows the corpus and test suite to evolve organically with the codebase through the fuzzer's feedback loop.

For a given commit or release under analysis, the standard behavior identifier executes the regression suite on the revised PUT and collects all distinct system call profiles observed during execution. A system call profile is the set of system call types triggered during the execution of an input, among those provided by the operating system API (e.g., `read`, `kill`, `open` in Linux). The resulting set of profiles provides a compact yet expressive characterization of the revised PUT's standard behaviors.

3.4 Vetting Changes for Backdoor Injections

When vetting a specific commit or release, the newer version of the PUT is fuzzed with a graybox fuzzer for a duration consistent with the target use case and practical constraints. In our evaluation, Lily uses a 10-minute fuzzing window with AFL++, which aligns with state-of-the-art CI fuzzing practices such as those used in CIFuzz. Longer fuzzing durations can be configured when CI runners allow it or when end users have sufficient computational resources to vet release updates more extensively.

Each input seed produced by the fuzzer during this vetting phase is analyzed in two steps. First, the seed is traced on the newer PUT, and its resulting system call profile is obtained. This profile is then compared to the set of standard profiles identified by the standard behavior identifier (Section 3.3) for the newer version of the PUT. If an exact match is found, the input is labeled *safe*. Second, if the profile does not match any standard behavior on the newer PUT, the input is traced again on the older version. If the same non-standard profile also appears there, the behavior is not caused by the vetted change, and the input is likewise labeled *safe*. Otherwise, the input is labeled *suspicious*, since the vetted change is responsible for introducing this new and atypical behavior—a characteristic frequently associated with backdoor injections [1, 28, 29]. Inputs classified as suspicious are forwarded to the suspicious code tracer to identify the code modifications responsible for the anomalous behavior.

A key aspect of this design is that the standard behavior identifier collects standard system call profiles **on the new version of the PUT**. This **helps mitigate false positives caused by benign changes** that legitimately add or remove system calls: the standard behaviors are always computed with these modifications already taken into account.

3.5 Producing Precise Backdoor Reports

When fuzzing the newer PUT reveals an input that triggers a non-standard system call profile, and this profile is not reproduced by executing the older PUT with the same input, Lily reports a backdoor injection attempt. To transform these findings into a more interpretable evaluation of suspicious code changes for inclusion in CI logs or release vetting reports, the suspicious code tracer proceeds as follows:

(1) **Isolate the backdoor trigger zone** by recording line coverage for the suspicious input within the modified code segments.
(2) **Isolate the backdoor impact zone** by using catchpoints for the suspicious system calls and backtraces to reveal the lines of code which lead to them (using GDB [11]).
(3) **Generate the final backdoor report**, which includes the trigger and impact zones (or their intersection, if present), along with a proof of concept (PoC): namely, the identified triggering input and the resulting non-standard system call profile.

3.6 Illustrative Example: The vsFTPd Backdoor

To illustrate how Lily operates on a real-world example, consider the vsFTPd backdoor [29] discussed earlier (Section 2.3.1).

When fuzzing the infected release of the program to determine whether any backdoor was introduced compared to the previous official release, the AFL++ fuzzer [9] eventually generates an input that triggers the backdoor—an FTP username containing the string `":)"`. Because this input activates the backdoor code, it exercises previously unseen control-flow edges; AFL++ therefore saves it as a seed and passes it to Lily for analysis. This seed produces a combination of system call types (shown in Listing 1) that never occur together when running the same infected release on all inputs from the existing regression suite—that is, the seeds AFL++ had accumulated while validating the earlier, non-infected release of vsFTPd. (Naturally, the fuzzer is unlikely to have saved the backdoor-triggering input among those earlier seeds, since the backdoor code did not exist in the previous release.)

Furthermore, the earlier, non-infected version of vsFTPd does not produce the same system call profile when executed on this input. Lily's backdoor detection oracle therefore concludes that:

(i) the input that triggers the backdoor induces a non-standard behavior in the infected release, and

```
--- a/sysdeputil.c
+++ b/sysdeputil.c
@@ -845,0 +847,23 @@
+int
+vsf_sysutil_extra(void)
+{
+  int fd, rfd;
+  struct sockaddr_in sa;
+  if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
+  exit(1);
+  memset(&sa, 0, sizeof(sa));
+  sa.sin_family = AF_INET;
+  sa.sin_port = htons(6200);
+  sa.sin_addr.s_addr = INADDR_ANY;
+  if((bind(fd,(struct sockaddr *)&sa,
+  sizeof(struct sockaddr))) < 0) exit(1);
+  if((listen(fd, 100)) == -1) exit(1);
+  for(;;)
+  {
+    rfd = accept(fd, 0, 0);
+    close(0); close(1); close(2);
+    dup2(rfd, 0); dup2(rfd, 1); dup2(rfd, 2);
+    execl("/bin/sh","sh",(char *)0);
+  }
+}
+
```

Listing 1: Example of a Lily finding report for the vsFTPd backdoor. The lines responsible for the suspicious system calls are marked in bold, and the C Library functions emitting these system calls are marked in red.

(ii) this behavior cannot be reproduced on the previous release.

Consequently, a backdoor is reported and the suspicious code tracer:

(1) collects the source line coverage for the suspicious input,
(2) intersects this coverage with the code differences between the two releases, and
(3) identifies the callsites responsible for the anomalous system calls, highlighting them within the code intersection from step (2).

As a result, the potentially suspicious modifications are narrowed down to a single file and seven lines, as shown in Listing 1. In contrast, the raw diff between the previous and infected releases spans five files and 1,407 lines of code. In the report, lines emitting suspicious system calls are displayed in **bold**, and the corresponding C library functions appear in **red**. This enables the human reviewer to quickly determine that the backdoor exposes a remote shell over a TCP socket, allowing an attacker to connect to the server and execute arbitrary commands. Based on this evidence, the reviewer can confidently confirm the presence of the backdoor, reject the entire release, and discard any artifacts derived from it that could otherwise be reused by Lily in subsequent iterations (e.g., seeds used to construct future regression test suites).

4 Mitigating Adversarial Attacks

As with existing CI fuzzing approaches, Lily adopts a best-effort strategy. In practice, it can detect and block the injection of real vulnerabilities. However, there are no hard guarantees that the fuzzer will trigger and identify all injected vulnerabilities. As with most security defenses, the primary objective is to *increase the cost of attacks* rather than make them entirely impossible. Nevertheless, if Lily is widely adopted, adversarial attackers may attempt to circumvent its mechanisms. In this section, we systematically review different attack scenarios, analyze their potential effectiveness, and discuss possible countermeasures.

4.1 Bypassing CI Checks

In many projects, core maintainers can bypass CI checks by adding a special command in the commit message or by using an option when pushing the commit to the repository. An attacker could exploit this to skip any Lily runs scheduled in the CI pipeline. To prevent this, Lily could run as an independent service provided by the Git hosting platform, ensuring that backdoor detection happens outside the control of the project's development team, and notifying all stakeholders should a backdoor be detected (e.g., by displaying a banner on the repository). This approach guarantees unbiased detection even if core maintainers are compromised [26].

4.2 Avoiding Backdoor Activation While Fuzzing

An attacker may attempt to conceal a backdoor in code that is difficult for a fuzzer to reach. To assess this risk, we measured reachable line coverage for 337 code-affecting commits across 12 open-source projects (see Section 5.1). Average final coverage is approximately 32%, although coverage varies substantially across commits: some are fully covered, whereas others receive only limited coverage. Our manual investigation revealed that low coverage primarily stems from limitations of the available developer-provided fuzzing harnesses. In several cases, the affected code resides in components that were not prioritized for fuzz testing, while in others it belongs to legacy functionality for which harnesses were still immature at the time. Although coverage is inherently constrained by available fuzzing resources, continued advances in computing performance and fuzzing techniques are likely to reduce these limitations over time. Increasing fuzzing budgets for randomly sampled Lily runs could further raise the cost of evasion trials. Regarding harnesses, we observed a consistent improvement in both quality and scope throughout the histories of the 12 projects we studied; when evaluated with modern harnesses, code-affecting commits typically achieve near-complete coverage. OSS-Fuzz maintainers indeed routinely refine existing harnesses, and recent advances in (semi-)automatic harness generation [5, 36] may further strengthen these efforts.

More advanced attacks may use anti-fuzzing techniques to prevent backdoor activation [19, 32], such as cryptographic checks like those in the XZ Utils incident [26]. However, CI and continuous fuzzing systems (e.g., OSS-Fuzz [2]) routinely generate coverage reports to help developers improve fuzzing harnesses and increase PUT coverage. Deliberately making parts of the PUT hard to reach could therefore backfire by drawing attention to newly uncovered code containing a backdoor. This may help explain why the attackers disabled CI fuzzing in the XZ Utils case.

4.3 Injecting the Backdoor in Multiple Steps

The PHP Attack. In the PHP incident [1], attackers tried to hide the backdoor by spreading it across multiple commits. The first commit (1) injected the backdoor, the second (2) reverted it (appearing to "fix" the issue), and a third (3) reverted the revert, restoring the original backdoor. We reproduced this scenario and found that

it had no impact on Lily. It detects the initial (1) and reverted (3) injections at a rate of 80%, while producing no false positives when the backdoor is removed in the first revert (2).

Injecting Benign System Call Arguments First. A more sophisticated attacker might try to bypass Lily by first adding a potentially dangerous piece of code with a benign argument (e.g., `system("echo a > /dev/null")`), later changing it to something malicious (e.g., `system("/bin/sh")`), hoping that Lily will treat system calls produced by `system()` as benign. However, this strategy would fail, as Lily flags the first use of `system()` as suspicious once it triggers a divergent system call profile, regardless of the argument. The attacker would then need to explain this addition to the codebase.

Modifying Preexisting System Calls. Going even further, an attacker might modify the arguments of a legitimate existing `system()` call. In this case, the system call profile is already part of representative behavior, so Lily considers it benign. Yet, to evade detection, the attacker must still ensure that the modified argument leads to the same system call profile. For instance, if `system("echo $USER > /var/connections.log")` is legitimate, an input reaching this code generates a system call profile including `clone`, `execve`, `open`, and `write`. Changing the code to `system("/bin/sh")` produces a different system call profile and triggers Lily. While it may be theoretically possible to craft an argument change that can reliably evade detection, Lily significantly raises the difficulty of a successful backdoor injection by limiting the attacker's options.

4.4 Reusing Common System Call Types

As a generalization of the previous attack, the attacker may design the backdoor to only reuse system call types already present in the PUT. Yet, to evade detection, the backdoor must be such that its activation *always* produces system call profiles identical to those of standard behaviors. Similar to the previous attack, this approach cannot be ruled out theoretically, yet appears practically challenging. Our evaluation includes several examples (see Section 5.1) where backdoors rely on system call types already present in the PUT, yet still fail to reliably evade detection.

4.5 Poisoning the Standard Behavior Corpus

A commit/release vetting campaign begins with Lily's standard behavior identifier, which collects seeds from the corpus inherited from the previous campaign, along with their corresponding system call profiles. These profiles represent the standard behaviors later used by Lily to detect deviations and report them as potential backdoors. An attacker may attempt to poison these standard behaviors with backdoor-compatible ones, preventing Lily from detecting a future backdoor injection. The attacker could indeed first embed the backdoor's trigger condition in the PUT while omitting the code that produces malicious behavior. During vetting, the fuzzer may save seeds that activate this trigger, which may persist in future sessions and create the poisoning effect.

Our experimental evaluation (Section 5.6) simulates the extreme scenario where an attacker succeeds in injecting 100 poisoned seeds without raising suspicion. This reduces Lily's backdoor detection rate, though it does not fully suppress it. To further mitigate this threat, we introduce a hardened mode for Lily, called LilySelective. In this mode, inputs from the inherited corpus producing divergent system call profiles between the old and new versions of the PUT are pruned. Our evaluation further indicates that this strategy systematically eliminates injected backdoor-compatible inputs, thereby blocking all poisoning attempts. However, it might also discard inputs that include legitimate behaviors, thereby increasing the likelihood of false alarms compared to vanilla Lily. This makes the two modes complementary for balancing safety with manual effort.

5 Experimental Evaluation

We aim at answering the following research questions:

RQ1 (backdoor detection rate) To what extent can Lily effectively prevent backdoor injections during time-constrained commit and release vetting, achieving detection rates sufficient for practical deployment?

RQ2 (a) (false alarm rate) To what extent is Lily sufficiently precise to be considered acceptable by project maintainers and end users, ensuring that it does not disrupt CI pipelines or new release vetting with frequent manual interventions caused by an excessive false-positive rate?

(b) (ablation study) To what extent does reporting a backdoor only for behaviors that are both novel and atypical reduce false alarm rates, thereby enhancing overall acceptability?

RQ3 (backdoor localization) To what extent does Lily generate precise backdoor reports, thereby enhancing acceptability?

RQ4 (corpus poisoning mitigation) How does the corpus poisoning attack in Section 4.5 reduce Lily's backdoor detection rate, and how effectively does the LilySelective hardened mode mitigate it?

5.1 Experimental Protocol

Tool Implementation and Overhead. Lily is implemented on top of AFL++ [9] (version `++4.34a`) for fuzzing, as AFL++ is actively maintained by the graybox fuzzing community and can be deemed representative of the current state of the art. Lily's oracle collects system calls through an additional tracing pass implemented with `strace` [39]. Detection of atypical behaviors is performed in parallel with fuzzing on a dedicated CPU core. Establishing the baseline of standard behaviors and flagging newly observed behaviors are performed before and after fuzzing, respectively, and each required less than one second in all our experiments.

Selecting Projects to Analyze. To select software projects for our experimental evaluation, we leverage Rosarum [21], the only existing dataset of (real and synthetic) code-level backdoors. Of its 17 real-world programs, we select 13 PUTs that are relevant to our backdoor injection detection scenario (see Table 1). Specifically, we only include open-source PUTs, as closed-source firmware lacks accessible source repositories and development history.[3] Consequently, we exclude the *Belkin*, *D-Link*, *Linksys*, and *Tenda* firmware components. We note, however, that Lily could be employed in the

[3]Although we could not locate any Git history for vsFTPd, we included it because source code for releases remains available.

Table 1: Relevant 13 programs from the ROSARUM benchmark used in our evaluation. Commit counts start at harness creation.

Name	Type	Commits	Backdoor description
PHP (**2021 attack**)	HTTP server	70 810 (since 2011)	HTTP request with secret field value enables command execution [1]
ProFTPD (**CVE-2010-20103**)	FTP server	12 680 (since 1998)	Secret FTP command leads to root shell [28]
vsFTPd (**CVE-2011-2523**)	FTP server	*No Git history*	FTP usernames containing ":)" lead to root shell [29]
libpng	Image library	543 (since 2017)	Secret image metadata values enables command execution
libsndfile	Sound library	376 (since 2019)	Secret sound file metadata value triggers home directory encryption
libtiff	Image library	1720 (since 2018)	Secret image metadata value enables command execution
libxml2	XML library	2264 (since 2021)	Secret XML node format enables command execution
Lua	Language interpreter	337 (since 2021)	Specific string values in script enables reading from filesystem
OpenSSL / bignum	Crypto library	20 479 (since 2016)	Secret bignum exponentiation string enables command execution
PHP / unserialize	Language interpreter	24 808 (since 2019)	Specific string values in serialized object enables PHP code execution
Poppler	PDF renderer	1523 (since 2020)	Secret character in PDF comment enables command execution
SQLite3	Database system	12 843 (since 2016)	Secret SQL keyword enables removal of home directory
Sudo	Unix utility	8354 (since 2010)	Hardcoded credentials

Table 2: Linux versions included in our release evaluation.

Distribution	Version	Codename	Launch date
Debian	11.0	bullseye	Aug. 2021
	12.0	bookworm	Jun. 2023
	13.0	trixie	Aug. 2025
Ubuntu	22.04 LTS	Jammy Jellyfish	Apr. 2022
	24.04 LTS	Noble Numbat	Apr. 2024
	25.04	Plucky Puffin	Apr. 2025

same manner internally by organizations developing closed-source software.

Selecting Commit and Release Pairs to Vet. In our evaluation, we sought to encompass the full development history of each of the 13 PUTs. However, this proved impractical due to the sheer volume of changes as well as the painstaking manual work required to adapt the build process, to ensure that all versions compile and fuzz correctly. For these reasons, we limited the evaluation of each project to the subset of the Git history for which a **fuzzing harness** is present. For **commit vetting**, we adopted the approach of Sharma et al. [35], sampling *representative committed changes* across the Git history. This process yields 18 commit pairs per PUT (affecting varying amounts of files and changing varying amounts of lines) to simulate a rolling CI job. Because some commits may not affect source code (e.g., documentation changes) or may not be coverable (e.g., comment additions), we repeated this process to select 18 additional pairs that guarantee source code changes and coverage, yielding a total of 36 commit pairs per PUT, with 432 pairs across all PUTs. For **release vetting**, we selected *four representative release pairs* per PUT by inspecting the releases included in three of the latest Ubuntu and Debian versions (see Table 2). Obtaining four pairs was not always possible, as identical PUT versions were sometimes reused across successive versions, yielding a total of 50 release pairs across all PUTs.

Finally, the commit and release pairs considered in our evaluation must fall into two categories: **legitimate changes** (to assess LILY's ability to identify them as harmless and avoid false alarms) and **backdoored changes** (to assess LILY's ability to detect injected backdoors). We manually vetted all 432 commit pairs and 50 release pairs; as expected, none contained backdoor injections, constituting exclusively legitimate changes. To incorporate backdoored changes, we implemented 13 commits by planting the ROSARUM backdoors in each corresponding PUT, and created variants of our 50 release pairs where the backdoor was present in the updated release versions.

In total, we obtained 545 version pairs, including three specifically built to **reproduce the ProFTPD, vsFTPd, and PHP attacks** described in Section 2.3.2. For PHP, we evaluate commit `c730aa26bd`, which corresponds to the first backdoor injection attempt. For ProFTPD and vsFTPd, we use the compromised releases from CVE-2010-20103 and CVE-2011-2523, where attackers replaced legitimate releases on the project servers.

Experimental Setup. For all version pairs, we follow standard fuzzing evaluation practices [33], and emulate typical CI resources. We use GitHub Action runners as a reference, allocating 4 CPU cores and 16 GiB of RAM per experiment. Each run lasts 10 minutes, matching common OSS-Fuzz settings,[4] and we repeat experiments 20 times to address fuzzing's non-determinism, for a total runtime of over 3600 CPU-hours. All tests run on a dedicated Intel® Xeon® Silver 4241 @ 2.20 GHz server. We adopt standard seed sets for LILY's initial seeds, minimized using AFL++'s standard corpus reduction. We release these seeds together with the paper's artifact.

5.2 RQ1: Backdoor Detection Rate

Table 3 summarizes LILY's backdoor detection rates across our 63 backdoored commits and releases. In a nutshell, all malicious changes were detected at least once across the 20 fuzzing runs. For **commit vetting**, LILY detected the backdoor in 90% of 260 runs. Among 13 PUTs, it achieved 100% detection for 6 (46%), at least 75% for 11 (85%), and at least 65% for all. For **release vetting**, LILY detected the backdoor in 83% of 1000 runs. Among 13 PUTs, detection was 100% for 7 (54%), at least 75% for 10 (73%), and at least 39% for all. This low detection rate for some PUTs can be attributed to the use of older, preliminary fuzzing harnesses; in the corresponding commit scenarios, the same backdoors are generally detected more frequently due to more mature and effective harnesses. Across 50 releases, the backdoor was detected in all runs for 30 (60%) and at least half for 40 (80%).

> ***Answer to RQ1*** (backdoor detection rate)
> Across 63 backdoor injection attacks—both real and synthetic—LILY achieved a **90% detection rate during commit vetting**

[4] Longer runs, up to 60 minutes, yielded a slight increase in detection rate with a marginally higher (yet still low overall) false positive rate.

Table 3: Backdoor detection rate of Lily on our backdoored commits and releases (RQ1).

Program	1 backdoored commit, 20 CI runs / commit	4 backdoored releases, 20 validation runs / release		
	Correctly blocked runs	# of releases: All runs block	# of releases: ≥ 1 runs block	Correctly blocked runs
PHP	16 / 20	0 / 4	4 / 4	61 / 80
ProFTPD	14 / 20	0 / 4	4 / 4	34 / 80 *
vsFTPd	20 / 20	3 / 3	3 / 3	60 / 60
libpng	16 / 20	0 / 4	4 / 4	31 / 80 *
libsndfile	13 / 20	0 / 4	4 / 4	32 / 80 *
libtiff	20 / 20	4 / 4	4 / 4	80 / 80
libxml2	18 / 20	3 / 3	3 / 3	60 / 60
Lua	19 / 20	4 / 4	4 / 4	80 / 80
OpenSSL / bignum	17 / 20	1 / 4	4 / 4	71 / 80
PHP / unserialize	20 / 20	4 / 4	4 / 4	80 / 80
Poppler	20 / 20	4 / 4	4 / 4	80 / 80
SQLite3	20 / 20	3 / 4	4 / 4	79 / 80
Sudo	20 / 20	4 / 4	4 / 4	80 / 80
TOTAL	**233 / 260**	**30 / 50**	**50 / 50**	**828 / 1000**

* Detection rates for ProFTPD, libpng, and libsndfile are lower in the release use case due to old releases relying on preliminary fuzzing harnesses.

and 83% during release vetting, averaged on 20 fuzzing trials **with only 10 minutes of fuzzing** per injection. In addition, **no injection remained undetected** across these trials, underscoring the robustness of the approach. These results highlight its practical relevance: had Lily been deployed at the time, **it could have automatically prevented three major incidents** affecting ProFTPD [28], vsFTPd [29], and PHP [1].

5.3 RQ2(a): False Alarm Rate

We evaluate Lily on the 432 commit pairs and 50 release pairs from our benchmark which represent legitimate changes, to measure its ability to classify them as harmless across 20 repeated vetting campaigns. Results are summarized in Table 4. For **commit vetting**, Lily raised false alarms in only 17 runs (0.2%) out of 8640. For 8 (67%) of the 12 relevant PUTs and 425 (98%) of the 432 commit pairs, no false alarms occurred. For **release vetting**, Lily reported false alarms in 43 runs (4.3%) out of 1000. For 9 (69%) of the 13 PUTs and 45 (90%) of the 50 release pairs, no false alarms occurred. Examining the few **code changes that triggered false alarms** provides valuable insights for potential users of Lily regarding rare scenarios that remain challenging. In PHP, four commit pairs caused false positives across 12 runs; all involved *merge commits* with massive, dispersed changes—patterns typical of major version releases. In libpng, a single commit pair introduced structural modifications to the fuzzing harness, naturally affecting detection outcomes. In ProFTPD, one commit pair exhibited "flaky" behavior [3], intermittently producing suspicious system calls and appearing in only one run. In Lua, one commit pair involved extensive memory management refactoring during a debug fix, altering system call patterns and leading to a false positive.

Answer to RQ2(a) (false alarm rate)

Across 482 valid code changes made to 13 programs, either at the single-commit level or between two releases, **Lily reported false alarms in only** 0.2% **during commit vetting and** 4.3% **during vetting of** typically much larger **release-level changes**. These results are averaged on 20 fuzzing trials with just 10 minutes of fuzzing per change. **Lily appears thus sufficiently reliable for automation**, incorrectly blocking fewer than one in 500 commits. Moreover, the few **code changes that trigger false alarms often exhibit recognizable patterns**, such as modifications to the fuzzing harness or merge commits. Overall, these findings suggest that Lily introduces a reasonable overhead compared to the manual effort required in traditional CI pipelines or new release testing processes.

5.4 RQ2(b): Ablation Study

Lily's backdoor detection oracle consists of two components (see Section 3.4): (1) a detector that flags new runtime behaviors (with regards to the pre-revision code), and (2) a detector that identifies runtime behaviors which significantly deviate from the revised code's typical execution profiles. Lily reports a backdoor only when *both* components raise an alert, allowing each to prune false positives produced by the other. To evaluate the contribution of each component, we replicate the experiments of RQ2(a) using two ablated variants of Lily: one that retains only the first component (**Novel Lily**) and one that retains only the second (**Atypical Lily**). We then compare both against the full tool (**Full Lily**).

Results are summarized in Table 4. For **commit vetting**, Atypical Lily produced false alarms in 1168 (14%) out of 8640 runs, Novel Lily 902 (10%), and Full Lily only 17 (0.2%). For **release vetting**, Atypical Lily produced false alarms in 199 (20%) out of 1000 runs, Novel Lily 581 (58%), and Full Lily only 43 (4.3%). **Overall**, these results show that every component of the Lily pipeline provides a significant contribution, collectively making Lily sufficiently automated for CI integration. In particular, while useful within the full pipeline, Novel Lily struggles to accommodate substantial changes across PUT versions. This limitation is amplified in release vetting, where Atypical Lily surpasses Novel Lily, as releases occur infrequently enough to accumulate many behavior changes between versions. All differences are **statistically significant**, with the Mann–Whitney U test performed across variants yielding $0.0003 < p < 0.05$, across both commits and releases.

Answer to RQ2(b) (ablation study)

Lily's detection oracle reports a backdoor only when *both* of its components raise an alert, enabling each component to prune the false positives produced by the other. Across 482 valid code changes on 13 programs, **the average false alarm rate dropped from 14% (commit) and 20% (release) when only atypical behaviors were flagged, and from 10% (commit) and 58% (release) when only new behaviors were flagged, to just** 0.2% **and** 4.3% when using the full Lily pipeline (20 runs of 10 minutes per code change). These results confirm that *both* components contribute significantly to the overall performance of the tool.

5.5 RQ3: Backdoor Localization

To evaluate Lily's ability to generate precise backdoor reports, we considered a worst-case scenario: a Linux distribution maintainer

Table 4: False alarm rate of Lily and its ablated variants on our valid commits and release updates (RQ2).

Program	Ablated variant	36 commits, 20 CI runs per commit			3 or 4 releases, 20 validation runs per release		
		# of wrongly blocked commits		# of wrongly blocked runs	# of wrongly blocked releases		# of wrongly blocked runs
		All runs block	≥ 1 runs block		All runs block	≥ 1 runs block	
PHP	(0) Atypical Lily	**0** / 36	34 / 36	232 / 720	**0** / 4	3 / 4	9 / 80
	(1) Novel Lily	9 / 36	22 / 36	262 / 720	3 / 4	4 / 4	77 / 80
	(2) **Lily** = (0) + (1)	**0 / 36**	**4 / 36**	**12 / 720**	**0 / 4**	**1 / 4**	**1 / 80**
ProFTPD	(0) Atypical Lily	2 / 36	36 / 36	393 / 720	3 / 4	4 / 4	78 / 80
	(1) Novel Lily	**0** / 36	4 / 36	19 / 720	2 / 4	4 / 4	50 / 80
	(2) **Lily** = (0) + (1)	**0 / 36**	**1 / 36**	**1 / 720**	**0 / 4**	**1 / 4**	**1 / 80**
vsFTPd	(0) Atypical Lily	*Git history not available*			**0** / 3	3 / 3	13 / 60
	(1) Novel Lily				**0** / 3	1 / 3	11 / 60
	(2) **Lily** = (0) + (1)				**0 / 3**	**0 / 3**	**0 / 60**
libpng	(0) Atypical Lily	**0** / 36	32 / 36	102 / 720	**0** / 4	3 / 4	14 / 80
	(1) Novel Lily	1 / 36	2 / 36	25 / 720	**0** / 4	4 / 4	33 / 80
	(2) **Lily** = (0) + (1)	**0 / 36**	**1 / 36**	**1 / 720**	**0 / 4**	**0 / 4**	**0 / 80**
libsndfile	(0) Atypical Lily	**0** / 36	33 / 36	81 / 720	**2** / 4	4 / 4	46 / 80
	(1) Novel Lily	**0** / 36	**0** / 36	**0** / 720	**2** / 4	**2** / 4	**40** / 80
	(2) **Lily** = (0) + (1)	**0 / 36**	**0 / 36**	**0 / 720**	**2 / 4**	**2 / 4**	**40 / 80**
libtiff	(0) Atypical Lily	**0** / 36	**0** / 36	**0** / 720	**0** / 4	**0** / 4	**0** / 80
	(1) Novel Lily	**0** / 36	2 / 36	10 / 720	3 / 4	3 / 4	60 / 80
	(2) **Lily** = (0) + (1)	**0 / 36**	**0 / 36**	**0 / 720**	**0 / 4**	**0 / 4**	**0 / 80**
libxml2	(0) Atypical Lily	**0** / 36	26 / 36	59 / 720	**0** / 3	2 / 3	2 / 60
	(1) Novel Lily	**0** / 36	1 / 36	14 / 720	1 / 3	1 / 3	20 / 60
	(2) **Lily** = (0) + (1)	**0 / 36**	**0 / 36**	**0 / 720**	**0 / 3**	**0 / 3**	**0 / 60**
Lua	(0) Atypical Lily	**0** / 36	7 / 36	25 / 720	**0** / 4	4 / 4	15 / 80
	(1) Novel Lily	2 / 36	6 / 36	70 / 720	**0** / 4	1 / 4	14 / 80
	(2) **Lily** = (0) + (1)	**0 / 36**	**1 / 36**	**3 / 720**	**0 / 4**	**0 / 4**	**0 / 80**
OpenSSL / bignum	(0) Atypical Lily	**0** / 36	**0** / 36	**0** / 720	**0** / 4	**0** / 4	**0** / 80
	(1) Novel Lily	**0** / 36	**0** / 36	**0** / 720	2 / 4	2 / 4	40 / 80
	(2) **Lily** = (0) + (1)	**0 / 36**	**0 / 36**	**0 / 720**	**0 / 4**	**0 / 4**	**0 / 80**
PHP / unserialize	(0) Atypical Lily	**0** / 36	**0** / 36	**0** / 720	**0** / 4	**0** / 4	**0** / 80
	(1) Novel Lily	25 / 36	26 / 36	502 / 720	4 / 4	4 / 4	80 / 80
	(2) **Lily** = (0) + (1)	**0 / 36**	**0 / 36**	**0 / 720**	**0 / 4**	**0 / 4**	**0 / 80**
Poppler	(0) Atypical Lily	**0** / 36	3 / 36	6 / 720	**0** / 4	**0** / 4	**0** / 80
	(1) Novel Lily	**0** / 36	**0** / 36	**0** / 720	3 / 4	3 / 4	60 / 80
	(2) **Lily** = (0) + (1)	**0 / 36**	**0 / 36**	**0 / 720**	**0 / 4**	**0 / 4**	**0 / 80**
SQLite3	(0) Atypical Lily	**0** / 36	36 / 36	270 / 720	**0** / 4	4 / 4	22 / 80
	(1) Novel Lily	**0** / 36	**0** / 36	**0** / 720	2 / 4	4 / 4	56 / 80
	(2) **Lily** = (0) + (1)	**0 / 36**	**0 / 36**	**0 / 720**	**0 / 4**	**1 / 4**	**1 / 80**
Sudo	(0) Atypical Lily	**0** / 36	**0** / 36	**0** / 720	**0** / 4	**0** / 4	**0** / 80
	(1) Novel Lily	**0** / 36	**0** / 36	**0** / 720	2 / 4	2 / 4	40 / 80
	(2) **Lily** = (0) + (1)	**0 / 36**	**0 / 36**	**0 / 720**	**0 / 4**	**0 / 4**	**0 / 80**
TOTAL	(0) Atypical Lily	2 / 432	207 / 432	1168 / 8640	5 / 50	27 / 50	199 / 1000
	(1) Novel Lily	37 / 432	63 / 432	902 / 8640	24 / 50	35 / 50	581 / 1000
	(2) **Lily** = (0) + (1)	**0 / 432**	**7 / 432**	**17 / 8640**	**2 / 50**	**5 / 50**	**43 / 1000**

receives a Lily report indicating a potential backdoor injection between two releases of an external package that differ by thousands to millions of lines of code. To instantiate this scenario, we injected the backdoor into the newer version of each pair of benchmarked releases, executed Lily, and examined the lines of code flagged by Lily's suspicious code tracer in the resulting backdoor report. Table 5 presents the averaged results. Overall, the suspicious code tracer dramatically reduces the maintainer's manual review burden, consistently shrinking the search space from thousands to millions of changed lines down to fewer than a dozen lines highlighted in Lily's backdoor reports. Manual inspection confirmed that all produced reports correctly pinpoint backdoor-revealing code, enabling the maintainer either to directly vet the suspicious changes or to raise a well-founded issue with the package developers.

Table 5: Size of Lily's reports during release vetting (RQ3).

Program	Avg. number of code lines in release diff	report
PHP	2,144,179	11 (0.0005%)
ProFTPD	120,944	12 (0.0099%)
vsFTPd	1589	6 (0.3776%)
libpng	34,113	4 (0.0117%)
libsndfile	47,408	5 (0.0105%)
libtiff	99,353	1 (0.0010%)
libxml2	132,294	2 (0.0015%)
Lua	5323	3 (0.0564%)
OpenSSL / bignum	625,018	2 (0.0003%)
PHP / unserialize	1,995,128	5 (0.0003%)
Poppler	151,602	3 (0.0020%)
SQLite3	345,731	3 (0.0009%)
Sudo	323,448	12 (0.0037%)
TOTAL	6,026,130	69 (0.0011%)

Answer to RQ3 (backdoor localization)

Lily generates **highly precise backdoor reports**. In our worst-case evaluation—where releases differed by thousands to millions of lines of code—the suspicious code tracer consistently narrowed the manual review effort to fewer than a dozen lines. **Every report correctly highlighted backdoor-revealing code**, enabling straightforward manual validation or escalation to release developers. This reliability and focus substantially enhance Lily's overall acceptability in real-world code-quality processes.

Table 6: Detection and false alarm rate of Lily and its hardened mode, LilySelective, using poisoned corpora (RQ4).

Tool mode	Commits, 20 CI runs per commit			Releases, 20 validation runs per release		
	# of blocked commits		# of blocked runs	# of blocked releases		# of blocked runs
	All runs block	≥ 1 runs block		All runs block	≥ 1 runs block	
Backdoor detection rate (13 **backdoored** commits, 50 **backdoored** releases, **poisoned** corpora—**more** blocked changes is better)						
Lily	3 / 13	9 / 13	140 / 260	15 / 50	34 / 50	455 / 1000
LilySelective	**6** / 13	**13** / 13	**233** / 260	**31** / 50	**50** / 50	**837** / 1000
False alarm rate (432 **safe** commits, 50 **safe** releases—**fewer** blocked changes is better)						
Lily	**0** / 432	**8** / 432	**18** / 8640	**0** / 50	**6** / 50	**35** / 1000
LilySelective	4 / 432	18 / 432	215 / 8640	9 / 50	11 / 50	182 / 1000

5.6 RQ4: Corpus Poisoning Mitigation

As discussed in Section 4.5, an attacker can directly or indirectly inject backdoor-triggering inputs into the standard behavior corpus, thereby poisoning a subsequent detection campaign. **To simulate this attack**, we repeat our experiments on the 545 legitimate or backdoored changes, but additionally inject 100 backdoor-triggering inputs (selected from those identified by Lily in the experiments of RQ1) to construct a *poisoned corpus* for each backdoored change. We run the backdoor detection campaigns using both the standard Lily tool and its hardened mode, LilySelective, described in Section 4.5. Table 6 reports the results. Poisoning almost halves Lily's detection rate, reducing it to 54% for commits and 46% for releases, but still maintaining a one-in-two chance of blocking the backdoor injection. In contrast, LilySelective completely nullifies the attack: it removes all poisoned inputs injected into the corpus and thus yields the same detection rates that Lily would achieve on a clean corpus. Yet, on some programs (such as ProFTPD), Lily's detection rate is entirely unaffected by the attack and is thus similar to that of LilySelective. This stems from the many distinct ways the backdoor can be triggered in these programs (i.e., producing different system call profiles), so that the 100 injected poisoned inputs are not sufficient to prevent detection by Lily. Finally, as expected, while Lily yields false-positive rates of 0.2% (commits) and 4% (releases), LilySelective exhibits higher rates of 3% and 18%, respectively.

Answer to RQ4 (corpus poisoning mitigation)
Our results over 63 backdoor injections indicate that **corpus poisoning attacks are feasible but intrinsically risky for an adversary**. Achieving a meaningful reduction in Lily's backdoor detection rate generally requires injecting a large number of diverse poisoned samples into the target corpus. Even then, success remains uncertain, and unsuccessful attempts are likely to expose the attacker. By contrast, **LilySelective hardened mode effectively neutralizes this attack vector**, though it introduces additional manual overhead for maintainers or end users, due to increased false alarms. To balance security and practicality, **periodic use of LilySelective should provide a strong deterrent**. Indeed, randomly auditing a subset of commits or releases in this mode significantly raises the attacker's risk of detection, while keeping the overall manual burden manageable.

5.7 Threats to Result Generalizability

As with all empirical research, this evaluation is subject to threats to generalizability. In the domain of code-level backdoor detection—and in particular the detection of *backdoor injections* in open-source repositories—these threats are especially pronounced. Documented injection incidents are rare, and authentic real-world samples remain difficult to obtain. This scarcity should not deter research efforts, however, as even a single successful compromise can have devastating consequences. The vsFTPd incident exemplifies this risk: had the attack succeeded, it could have granted its operator unauthorized access to millions of devices worldwide.

Given this context, evaluation methodologies commonly used for frequent but moderate-impact vulnerabilities (such as C memory bugs, which are often not exploitable [24]) must be adapted to suit rarer but extremely high-impact threats. Our evaluation follows state-of-the-art recommendations for fuzzing experiments [33] and relies on a backdoor detection benchmark that we introduced at ICSE'25. The benchmark and its construction methodology are described in detail in our ICSE paper [21]. Overall, we evaluate on 13 backdoor attacks: three real-world attacks (including two high-impact CVEs that remained undetected for several days) and ten synthetic but realistic ones, all affecting widely used open-source projects. While evaluating Lily on live open-source projects would provide valuable insights into its real-world applicability and impact, we leave this direction to future work, as it requires collaboration with project maintainers and long-term observation over years.

Additional threats to the generalizability of the results arise from the variety of code changes, program types, and adversarial strategies that may fall outside our evaluation scope. To mitigate these concerns, our experiments cover 545 distinct code changes—ranging from single commits to major multi-year releases—across all 13 diverse open-source projects in the benchmark. In Section 4, we further examine several plausible attacker strategies aimed at evading Lily, discuss qualitative and quantitative countermeasures, and acknowledge current limitations.

6 Related Work

Preventing Backdoor Injections. Ganz et al. [12] train a machine-learning model to flag anomalous contributor behavior indicative of malicious injections in Git histories. Because it focuses on developer activity rather than code, their approach is orthogonal and complementary to Lily. We considered evaluating it, but practical and methodological issues prevented a meaningful comparison. Although an implementation exists, it is undocumented and could not be executed on a modern system, and the authors did not respond

to inquiries. Moreover, their 19 benchmarks largely fall outside our scope: only one (PHP) aligns with our setting; many involve student projects infected with the same worm, purely destructive attacks, or languages with limited fuzzing support. On the shared PHP benchmark, Ganz et al. report an 8.25% false-alarm rate, whereas Lily raises none. Across all benchmarks, their method averages 10.75% false positives while detecting 15 (79%) attacks. In contrast, Lily yields 0.18% false positives over 432 clean commit pairs and detects all 13 injected backdoors. Execution time and resource usage are unreported, preventing assessment of CI feasibility.

Reproducible Builds (R-B) [25] strengthen the software supply chain by ensuring that independent builds of the same source yield bit-for-bit identical binaries, allowing detection of tampering during build or distribution. R-B complements Lily, which detects backdoor injections at commit time and reveals backdoors hidden in dependencies—threats reproducible builds alone cannot address. Combining Lily with R-B's static verification to compare a new release against a trusted prior version provides two independent and mutually reinforcing checks against out-of-repository injections and build-time manipulation.

Detecting Code-Level Backdoors in Binaries. Automated tools for identifying pre-existing code-level backdoors target off-the-shelf binary components and include only five proposals in the last decade.

Our Rosa [21] approach shares some technical components with Lily, namely graybox fuzzing and system call tracing. However, the two approaches are based on substantially different principles. Rosa performs two fuzzing campaigns on the same binary-only PUT: a short campaign and a long campaign. A potential backdoor is reported when a divergence in system-call behavior is observed between the two campaigns. The underlying intuition is that, if the short campaign is sufficiently brief, the fuzzer is unlikely to discover and trigger a hidden backdoor. In practice, however, the duration of the short campaign must be carefully calibrated for each PUT, fuzzing configuration, and computational environment. Furthermore, our experiments reported in the Rosa paper [21], conducted on the same benchmark and hardware used to evaluate Lily, indicate that Rosa typically generates several false positives per campaign that must be manually triaged and often requires several hours to detect a backdoor. While these limitations are acceptable in the lengthy vetting cycles of embedded firmware—the setting for which Rosa was originally designed—they become problematic in Lily's target CI/release environment. In this setting, analyses must complete automatically within roughly ten minutes and must not disrupt development or release workflows with false alarms. Consequently, high precision is essential for maintaining developer productivity and preserving maintainer trust. That said, software projects deploying Lily for the first time could use Rosa to verify the absence of pre-existing backdoors, thereby establishing a clean baseline before enabling continuous monitoring.

All four remaining approaches require significant manual effort—either to filter out false positives or to reverse-engineer suspicious or sensitive code—rendering them equally unsuitable for CI and release pipelines. Weasel [34] dynamically analyzes execution traces to locate command and authentication handlers, but still requires substantial manual review. HumIDIFy [46] uses machine learning to infer implemented protocols and checks compliance against human-written high-level feature lists. Firmalice [37] relies on symbolic execution to trigger sensitive operations without authentication, but depends on manually chosen target functions. Stringer [44] performs static analysis to extract suspicious string constants, often generating hundreds of false positives.

Fuzzing in a CI Context. Fuzzing is widely used in CI, largely through OSS-Fuzz [2]'s *CIFuzz* [31, 40–42]. Recent work studies directed CI fuzzing [13, 17, 35] and compiler-level fuzzing [4]. Lily targets a new threat class—backdoors—and shows they can be detected under CIFuzz-like constraints: short runs, limited resources, and minimal false alarms, enabling seamless integration.

Automated Regression Testing. Prior work on regression test generation [18, 22] has investigated the automatic construction of test suites that expose behavioral differences between software versions, typically by comparing internal states, return values, or program outputs. Adapting these techniques to identify inputs that induce system call–level differences, and integrating them with Lily's fuzzing backend, could guide the fuzzer toward regions of the input space that are more likely to exhibit suspicious new behaviors and therefore be flagged by Lily.

7 Conclusion

In this work, we have introduced Lily, a new approach that significantly strengthens the security of open-source development and release workflows by automatically detecting code-level backdoors before they reach users. Lily enables fast, automated, and precise identification of malicious code changes at scale. Evaluation across diverse benign and malicious commits shows that Lily achieves high accuracy (90% detection rate on average in the commit scenario, 83% in the release scenario), maintains low false alarm rates (0.2% on average for commits, 4.3% for releases), and remains robust against adversarial evasion strategies. Our experiments also reveal that Lily would have prevented multiple real-world backdoor incidents and thus offers a practical path toward safer open-source software ecosystems.

Data Availability

Lily is available at https://github.com/binsec/rosa/tree/lily and archived on Software Heritage with SWHID swh:1:rev:2bd97b1c06c315a986d969fddec6049b1cc27118.
A result replication package is available at https://zenodo.org/records/19337349.

Acknowledgments

This work was supported by the French National Research Agency (*Agence Nationale de la Recherche*, ANR) under the JCJC program (ANR-22-CE39-0012-01) and the France 2030 initiative (ANR-22-PTCC-0001 / SECUBIC).

References

[1] 2021. PHP 1.8.0-dev Backdoor. https://news-web.php.net/php.internals/113838.

[2] Abhishek Arya, Oliver Chang, Jonathan Metzman, Kostya Serebryany, and Dongge Liu. [n. d.]. *OSS-Fuzz*. https://github.com/google/oss-fuzz.

[3] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. D ᴇ F Lᴀᴋᴇʀ: Automatically Detecting Flaky Tests. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 433–444. doi:10.1145/3180155.3180164

[4] Karnbongkot Boonriong, Stefan Zetzsche, and Alastair F. Donaldson. 2025. Compiler Fuzzing in Continuous Integration: A Case Study on Dafny. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 441–452. doi:10.1109/ICST62969.2025.10988954

[5] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. 2023. Hopper: Interpretative Fuzzing for Libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Copenhagen Denmark, 1600–1614. doi:10.1145/3576915.3616610

[6] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. 2017. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526* (2017). doi:10.48550/arXiv.1712.05526

[7] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empir. Softw. Eng.* 24, 1 (2019), 381–416. doi:10.1007/S10664-017-9589-Y

[8] Yong Fang, Mingyu Xie, and Cheng Huang. 2021. PBDT: Python Backdoor Detection Model Based on Combined Features. *Security and Communication Networks* 2021 (Sept. 2021), 1–13. doi:10.1155/2021/9923234

[9] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Mark Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *WOOT'20: Proceedings of the 14th USENIX Conference on Offensive Technologies*. 10.

[10] Martin Fowler. 2024. Continuous Integration. https://www.martinfowler.com/articles/continuousIntegration.html.

[11] Free Software Foundation. 2025. *The GNU Project Debugger*. https://www.sourceware.org/gdb/.

[12] Tom Ganz, Inaam Ashraf, Martin Härterich, and Konrad Rieck. 2023. Detecting Backdoors in Collaboration Graphs of Software Repositories. In *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*. ACM, Charlotte NC USA, 189–200. doi:10.1145/3577923.3583657

[13] Elia Geretto, Andrea Jemmett, Cristiano Giuffrida, and Herbert Bos. 2025. LibAFLGo: Evaluating and Advancing Directed Greybox Fuzzing. In *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*. IEEE, Venice, Italy, 355–373. doi:10.1109/EuroSP63326.2025.00029

[14] Git. 2025. *Git version control system*. https://github.com/git/git.

[15] Patrice Godefroid. 2020. Fuzzing: Hack, Art, and Science. *Commun. ACM* 63, 2 (Jan. 2020), 70–76. doi:10.1145/3363824

[16] Petr Hosek and Cristian Cadar. 2015. VARAN the Unbelievable: An Efficient N-version Execution Framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 339–353. doi:10.1145/2694344.2694390

[17] Madonna Huang and Caroline Lemieux. 2024. Directed or Undirected: Investigating Fuzzing Strategies in a CI/CD Setup (Registered Report). In *Proceedings of the 3rd ACM International Fuzzing Workshop*. ACM, Vienna Austria, 33–41. doi:10.1145/3678722.3685532

[18] Wei Jin, Alessandro Orso, and Tao Xie. 2010. Automated Behavioral Regression Testing. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST '10)*. IEEE Computer Society, USA, 137–146. doi:10.1109/ICST.2010.64

[19] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. 2019. Fuzzification: Anti-Fuzzing Techniques. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1913–1930. https://www.usenix.org/conference/usenixsecurity19/presentation/jung.

[20] Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben Ten Hove, and Marcel Böhme. 2023. Continuous Fuzzing: A Study of the Effectiveness and Scalability of Fuzzing in CI/CD Pipelines. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. IEEE, Melbourne, Australia, 25–32. doi:10.1109/SBFT59156.2023.00015

[21] Dimitri Kokkonis, Michaël Marcozzi, Emilien Decoux, and Stefano Zacchiroli. 2025. ROSA: Finding Backdoors with Fuzzing. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2816–2828. doi:10.1109/ICSE55347.2025.00183

[22] Bogdan Korel and Ali M. Al-Yami. 1998. Automated regression test generation. *SIGSOFT Softw. Eng. Notes* 23, 2 (March 1998), 143–152. doi:10.1145/271775.271803

[23] Nadiya Kostyuk and Susan Landau. 2022. Dueling Over Dual_EC_DRBG: The Consequences of Corrupting a Cryptographic Standardization Process. *Harv. Nat'l Sec. J.* 13 (2022), 224.

[24] Guilhem Lacombe and Sébastien Bardin. 2025. Attacker control and bug prioritization. In *Proceedings of the 34th USENIX Conference on Security Symposium* (Seattle, WA, USA) *(SEC '25)*. USENIX Association, USA, Article 234, 20 pages.

[25] Chris Lamb and Stefano Zacchiroli. 2022. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software* 39, 2 (2022), 62–70. doi:10.1109/MS.2021.3073045

[26] Mario Lins, René Mayrhofer, and Michael Roland. 2025. Unveiling the Critical Attack Path for Implanting Backdoors in Supply Chains: Practical Experience from XZ. In *Cryptology and Network Security: 24th International Conference, CANS 2025, Osaka, Japan, November 17–20, 2025, Proceedings* (Osaka, Japan). Springer-Verlag, Berlin, Heidelberg, 521–541. doi:10.1007/978-981-95-4434-9_24

[27] Vaibhav G. Lokhande and Deepti Vidyarthi. 2019. A study of hardware architecture based attacks to bypass operating system security. *Security and Privacy* 2, 4 (June 2019), 11 pages. doi:10.1002/spy2.81

[28] NVD NIST. 2010. CVE-2010-20103. https://nvd.nist.gov/vuln/detail/CVE-2010-20103.

[29] NVD NIST. 2011. CVE-2011-2523. https://nvd.nist.gov/vuln/detail/CVE-2011-2523.

[30] NVD NIST. 2024. CVE-2024-3094. https://nvd.nist.gov/vuln/detail/CVE-2024-3094.

[31] OpenSSL. 2025. *OpenSSL*. https://github.com/openssl/openssl.

[32] Vasil Sarafov, David Markvica, and Stefan Brunthaler. 2025. Tephra: Principled Discovery of Fuzzer Limitations. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2733–2745. doi:10.1109/ASE63991.2025.00224

[33] 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, San Francisco, CA, USA, 1974–1993. doi:10.1109/sp54263.2024.00137

[34] Felix Schuster and Thorsten Holz. 2013. Towards Reducing the Attack Surface of Software Backdoors. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security - CCS '13*. ACM Press, Berlin, Germany, 851–862. doi:10.1145/2508859.2516716

[35] Arindam Sharma, Cristian Cadar, and Jonathan Metzman. 2024. Effective Fuzzing within CI/CD Pipelines (Registered Report). In *Proceedings of the 3rd ACM International Fuzzing Workshop*. ACM, Vienna Austria, 52–60. doi:10.1145/3678722.3685534

[36] Gabriel Sherman and Stefan Nagy. 2025. No Harness, No Problem: Oracle-guided Harnessing for Auto-generating C API Fuzzing Harnesses. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, Ottawa, ON, Canada, 165–177. doi:10.1109/ICSE55347.2025.00239

[37] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. doi:10.14722/ndss.2015.23294

[38] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1275–1295. doi:10.1109/SP.2019.00010

[39] Strace. 2025. *Strace Linux utility*. https://github.com/strace/strace.

[40] Sudo Project. 2025. *Sudo*. https://www.sudo.ws/.

[41] The libsndfile team. 2025. *libsndfile*. https://github.com/libsndfile/libsndfile.

[42] The PHP Foundation. 2025. *PHP*. https://github.com/php/php-src.

[43] The PNG Development Group. 2025. *libpng*. https://github.com/pnggroup/libpng.

[44] Sam L. Thomas, Tom Chothia, and Flavio D. Garcia. 2017. Stringer: Measuring the Importance of Static Data Comparisons to Detect Backdoors and Undocumented Functionality. In *Computer Security – ESORICS 2017*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Vol. 10493. Springer International Publishing, Cham, 513–531. doi:10.1007/978-3-319-66399-9_28

[45] Sam L. Thomas and Aurélien Francillon. 2018. Backdoors: Definition, Deniability and Detection. In *Research in Attacks, Intrusions, and Defenses*, Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis (Eds.). Vol. 11050. Springer International Publishing, Cham, 92–113. doi:10.1007/978-3-030-00470-5_5

[46] Sam L. Thomas, Flavio D. Garcia, and Tom Chothia. 2017. HumIDIFy: A Tool for Hidden Functionality Detection in Firmware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Michalis Polychronakis and Michael Meier (Eds.). Vol. 10327. Springer International Publishing, Cham, 279–300. doi:10.1007/978-3-319-60876-1_13

Received 2026-03-25; accepted 2026-06-18