

Steering Directed Kernel Fuzzing Beyond Reachability Filtering with Attribution Guidance

Zhiyu Zhang^{1,2}, Longxing Li^{1,2}, Ruigang Liang^{1,2,*}, Kai Chen^{1,2,*}, Junyan Jiang^{1,2}, Chen Liang³, Changwei Xu⁴

¹Institute of Information Engineering, Chinese Academy of Sciences

²School of Cyber Security, University of Chinese Academy of Sciences

³School of Artificial Intelligence, UCAS ⁴The University of Hong Kong

{zhangzhiyu1999, lilongxing, liangruigang, chenkai, jiangjunyan}@iie.ac.cn
liangchen24@mailsucas.ac.cn, changwei_xu@connect.hku.hk

Abstract—Directed greybox fuzzing is pivotal for bug re-exposure and patch testing, aiming to steer execution toward specific target sites. However, existing directed kernel fuzzers struggle to balance precision with efficiency. Distance-based approaches rely heavily on computationally expensive static analysis and iterative compilation, incurring significant pre-fuzzing overhead. Conversely, learning-based input pruning methods, while successful in user-space, lack adaptability in the kernel domain. Current models either optimize general coverage or are trained offline on fixed datasets, which limits their reuse when the target site, kernel version, or configuration changes.

In this paper, we present SYZPILOT, a target-adaptive framework that steers directed kernel fuzzing via online reachability prediction and attribution-guided scheduling. SYZPILOT uses a domain-specific encoder and a curriculum learning strategy to learn target-specific progress from runtime feedback and to prioritize inputs according to intermediate waypoints. To bridge the gap between reachability and bug exposure, SYZPILOT leverages attribution scores from its reachability predictor and recurring syscall patterns observed in target-reaching executions to guide mutation and increase opportunities for bug exposure. Our evaluation demonstrates that SYZPILOT outperforms state-of-the-art fuzzers by reproducing $1.71\text{--}5.14\times$ more bugs. Specifically, it achieves $6.87\text{--}261.26\times$ log1p-scale improvements in target-hit executions and $1.47\text{--}1.83\times$ speedups in bug exposure, respectively. SYZPILOT also found and reported 5 unknown bugs with one CVE assigned.

I. INTRODUCTION

As the Linux kernel underpins billions of devices, vulnerabilities within its code pose severe security risks, ranging from system crashes to privilege escalation. Among many static analysis and dynamic testing studies [1], [2], [3], [4], [5], [6], coverage-guided fuzzing has proven the most effective in practice. Syzkaller [7], a coverage-guided kernel fuzzer, attempts to extract and minimize a reproducer by replaying logged inputs in a clean VM, but kernel state or temporal

dependencies may prevent the crash from recurring. In practice, syzbot [8], powered by Syzkaller, had reported 28.4k bug reports as of August 2026, of which 19.4k (68.20%) lacked both C and syz reproducers. The absence of a stable reproducer hinders developers from analyzing root causes and testing patches. Re-exposing the reported bug can still provide a fresh failure execution and associated crash artifacts for diagnosis or further replay and minimization attempts. In this paper, “bug reproduction” means re-exposing the target crash rather than generating a stable reproducer. This focus complements research on PoC generation [9], [10]. Directed exploration can also seek runtime failure evidence at candidate locations identified by static or LLM-assisted code analysis [11], [12]. This motivates directed greybox fuzzing (DGF), which steers mutation and scheduling toward specific target sites (e.g., crash locations or patched code) to improve bug reproduction, patch testing, and target-driven vulnerability discovery. In user-space software, DGF typically follows two categories: *distance-based minimization* and *reachability-based pruning* [13]. However, adapting these strategies to the OS kernel is non-trivial, and renders existing approaches inefficient.

State-of-the-art Directed Kernel Fuzzers (DKFs) predominantly adopt the *distance-minimization* paradigm combined with syscall inference analysis [14], [15], with per-target static analysis and instrumentation. For instance, SyzDirect [15] uses a “compile-analyze-recompile” workflow: compiling the kernel to bitcode, performing complex inter-procedural analysis to extract syscall entry mappings and calculate distances, and finally recompiling for instrumentation. This pre-processing often requires tens of minutes of manual configuration and tens of CPU hours for a single target, which is costly when repeated across many targets or kernel versions. These analyzers are also tied to kernel and compiler representations assumed by their LLVM passes. Changes in struct layouts, function inlining, or ConstExpr and GEP forms can cause missed handlers or analysis failures on unsupported builds, although distance scheduling remains effective when preprocessing succeeds. Similarly, due to the existence of indirect calls, nested data structures, and path explosion, adapting precondition-analysis-based pruning methods (e.g., Beacon [16]) to the complex kernel context is also infeasible.

*Corresponding authors.

Conversely, the learning-based *reachability-based pruning* approach (e.g., FuzzGuard [17]), which uses machine learning to filter inputs predicted to miss the target, remains largely unexplored in the kernel domain. Unlike user-space input formats, kernel inputs involve complex, state-dependent syscall sequences that require semantic awareness. Existing neural models for kernel fuzzing [18], [19], [20], such as Snowplow [20], only utilize Language Models or Graph Neural Networks (GNNs) for general code coverage. They lack the specificity to target distinct sites or rely on offline training with fixed datasets. Consequently, these models do not provide target-specific adaptation across campaigns whose target site, kernel build, or configuration changes.

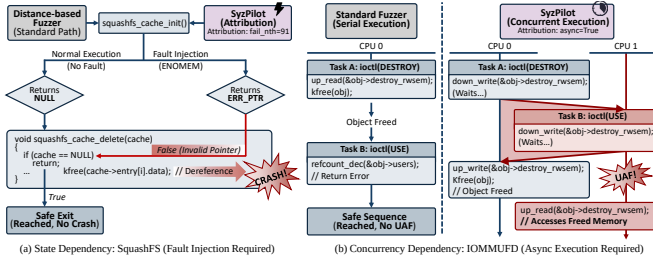


Fig. 1. The gap between reachability and bug exposure.

These limitations suggest that directed kernel fuzzing needs a different form of target guidance. Static distance analysis provides target specificity, but its per-target preprocessing is costly to repeat and tied to kernel and compiler representations. Existing learning-based pruning avoids such preprocessing, but it is not target-adaptive and does not model syscall-sequence semantics. We therefore explore a third direction: learning target-specific progress from runtime coverage collected during fuzzing. However, learning whether an input reaches target code is still not enough. Reaching a target basic block does not imply exposing the bug. The missing link is the kernel context and state, such as fault injection, thread interleaving, resource state, or syscall ordering, which moves an execution from target reachability to bug exposure. Consider the real-world bugs in Figure 1. Figure 1(a) depicts a state-dependent bug in the `squashfs`. A standard distance-based fuzzer may successfully reach the cleanup function `squashfs_cache_delete`. However, without a specific fault injection into `squashfs_cache_init`, the function returns NULL rather than an error pointer, causing the vulnerable check to exit safely. Similarly, Figure 1(b) shows a concurrency-dependent UAF in `iommufd`. While a fuzzer might cover the requisite syscalls, it may miss the narrow race context. In both cases, optimizing only for reachability is insufficient because the fuzzer treats the target as a scalar endpoint and receives no guidance about the execution properties that make the bug observable. Realizing a target-adaptive and exposure-aware DKF faces three challenges:

Challenges. C1: Low-cost Target-specific Progress Guidance. A directed kernel fuzzer needs a progress signal that is more informative than undirected coverage and less tied to kernel-

and compiler-specific preprocessing. The challenge is to obtain staged guidance toward a target location without requiring repetitive “compile-analyze-recompile” preprocessing for every target. **C2: Cold-started Online Learning with Imbalanced Data.** To adapt to target-specific dynamics, the reachability classifier should be trained online. However, directed targets are often deeply nested and rarely visited initially. This creates a severe class imbalance where positive samples are extremely scarce during the cold-start phase. Training a classifier on such skewed data can cause promising inputs to be misclassified as unreachable and skipped before the fuzzer gains traction. **C3: The Reachability-to-Exposure Gap.** Reaching a target basic block is necessary but not sufficient for reproducing a kernel bug. Bug exposure often depends on execution context, such as fault injection, thread interleavings, resource states, or syscall ordering. A reachability predictor alone provides no direct guidance on input mutation toward bug-triggering contexts.

SYZPILOT. To address these challenges, we propose SYZPILOT, a target-adaptive framework that steers directed fuzzing via reachability prediction and attribution-guided scheduling. First, to provide low-cost target-specific progress guidance (C1), SYZPILOT extracts intermediate waypoints from available debugging context, such as call traces in Syzkaller-style bug reports, and uses them as staged runtime labels. When only a target site is available, SYZPILOT can optionally derive less restrictive graph-based waypoints from a matching kernel call graph for target reaching. Second, to handle cold-started online learning with skewed waypoint labels (C2), SYZPILOT combines lightweight cold-start syscall inference, a pre-trained SyzEncoder, and adaptive curriculum training. The Path Analyzer biases early generation toward syscalls likely to interact with the target subsystem, while SyzEncoder provides semantic representations of structured syscall sequences for effective online learning. The Curriculum Trainer first groups waypoint labels into coarse classes, then introduces waypoint-level classification as positive samples accumulate. Finally, to bridge the reachability-to-exposure gap (C3), SYZPILOT uses attribution-guided scheduling. It combines model-based syscall and property attribution with sequence mining over target-reaching programs to prioritize mutations involving the observed syscalls, execution properties, and sequence templates and increase opportunities for bug exposure.

SYZPILOT features a distributed architecture where inference and training are decoupled from the fuzzer, reducing interference with fuzzing throughput. We implemented SYZPILOT based on Syzkaller (6e83b42) and evaluated it against Snowplow, SyzDirect, and Syzkaller on a 70-bug benchmark with ten runs per target. Under the standard 24-hour fuzzing budget, SYZPILOT reaches 57 targets and reproduces 36 bugs, increasing target-hit executions by $6.87\text{--}261.26\times$ (log1p-scale) and speeding up bug exposure by $1.47\text{--}1.83\times$.

Contributions. We summarize our contributions as follows:

- We propose SYZPILOT, a target-adaptive directed kernel fuzzing framework that combines online curriculum learning for input pruning with attribution-guided scheduling for bug exposure.

- We release a pre-trained kernel-aware language model SyzEncoder with corresponding dataset to facilitate future neural kernel fuzzing research. We also open-source SYZPILOT to facilitate replication.

- We evaluate SYZPILOT on a 70-bug benchmark and show that it reproduces $1.71\text{--}5.14\times$ more bugs and achieves $6.87\text{--}261.26\times$ log1p-scale improvements in target-hit executions and accelerates bug exposure by $1.47\text{--}1.83\times$.

II. BACKGROUND AND RELATED WORK

A. Directed Greybox Fuzzing

Directed Greybox Fuzzing (DGF) aims to focus testing on predefined target locations, such as potential vulnerabilities, specific patches, or critical code segments, rather than exploring the program indiscriminately. Generally, DGF strategies fall into the following two primary categories.

Distance-based Minimization. Pioneered by AFLGo [21], these approaches guide the fuzzer by minimizing the distance between the current execution path and the target sites on the CFG. Subsequent work optimized this strategy: Hawkeye [22] and WindRanger [23] introduce finer-grained distance metrics and prioritized scheduling. DAFL [24] incorporates data-flow analysis to refine the reachability distance. More recently, DeepGo [25] models DGF as a path transition process, utilizing reinforcement learning to generate optimal transition sequences, while IDFuzz [26] employs neural networks to learn mutation strategies from historical inputs.

Reachability-based Pruning. Such approaches improve efficiency by filtering out inputs predicted or statically determined to miss the target. FuzzGuard [17] employs a Convolutional Neural Network (CNN) to predict reachability, pruning ineffective inputs before execution. SieveFuzz [27] utilizes lightweight forward control-flow analysis to identify and block unreachable paths via tripwires. Similarly, Beacon [16] leverages static interval analysis and preconditions to prune execution paths that cannot theoretically reach the target. In addition to distance scheduling, WAFLGo [28] conducts targeted testing on commit code by guiding input generation.

Fine-Grained Coverage Feedback. Beyond code coverage, state and data-flow coverage feedback distinguish executions by program states or influential input-variable relations [29], [30], while domain-specific feedback prioritizes selected behaviors [31]. WingFuzz and StorFuzz retain inputs with diverse data interactions [32], [33]. These signals could complement directed guidance, but none, to our knowledge, has been evaluated for directed kernel bug reproduction.

While effective in user-space, these approaches face a semantic gap when applied to the OS kernel. Distance-based methods rely on accurate CFG extraction, which is notoriously difficult in kernels due to indirect calls and interrupt handlers. Furthermore, input pruning tools like FuzzGuard operate on raw byte streams, failing to handle highly structured, syntax-dependent system call sequences inherent to kernel fuzzing.

TABLE I
COMPARISON OF SYZPILOT WITH STATE-OF-THE-ART DIRECTED FUZZING AND KERNEL FUZZING APPROACHES.

Tool	Domain	Directed Strategy	Lightweight Static Analysis	Online Learning	Attribution Guidance
<i>User-Space DGF</i>					
AFLGo [21]	User-Space	Distance	○	○	○
FuzzGuard [17]	User-Space	Pruning	○	●	○
Beacon [16]	User-Space	Pruning	○	○	○
<i>Kernel DGF</i>					
SyzDirect [15]	Kernel	Distance	○	○	●
G-Fuzz [14]	Kernel	Distance	○	○	●
SyzLego [34]	Kernel	Distance	○	○	○
DDRace [42]	Kernel	Bug-Specific	○	○	●
<i>Learning-Aided Kernel Fuzzing</i>					
Snowcat [18]	Kernel	Non-directed	○	○	○
Snowplow [20]	Kernel	Non-directed	○	○	○
MOCK [19]	Kernel	Non-directed	●	●	○
SYZPILOT	Kernel	Pruning+Attribution	●	●	●

B. Directed Kernel Fuzzing

Recent research has adapted DGF to the kernel domain, broadly categorized into static-analysis-based guidance and vulnerability-specific guidance.

Static-Analysis-based Guidance. SyzDirect [15] is the state-of-the-art representative, which identifies entry system calls via static analysis and instruments the kernel to calculate basic-block distances. G-Fuzz [14] enhances this with finer-grained distance calculation and target-related syscall inference. SyzLego [34] further assists DGF by inferring syscall dependencies through static analysis to adjust task scheduling. DRLF-C [35], [36], [37] explores DKF in industry scenarios. **Vulnerability-Specific Guidance.** Several tools optimize for specific bug types [38], [39] rather than generic code reachability. GREBE [40] focuses on exploring error handling code to discover more bugs from existing PoCs. SyzScope [41] aims to escalate low-risk bugs to high-risk exploits. DDRace [42] extracts race pairs to target race conditions. CountDown [43] utilizes dynamic refcount operations to reshape syscall relationships.

As summarized in Table I, current directed kernel fuzzers still leave two gaps for practical bug reproduction. First, distance-based systems rely on per-target static analysis and instrumentation, which can be costly to repeat across many reported bugs and tied to kernel- and compiler-specific preprocessing. Second, vulnerability-specific methods provide strong guidance for particular bug classes, but they do not offer a general mechanism for target-specific adaptation across campaigns or bridging reachability and bug exposure.

C. Deep Learning and LLMs for Kernel Fuzzing

Recent work has integrated Deep Learning (DL) and Large Language Models (LLMs) to enhance kernel fuzzing. SyzVe-gas [44] applies multi-armed bandit reinforcement learning to dynamically select seeds and mutation tasks. MOCK [19] trains a language model to learn context-aware dependencies from syscall sequences. Snowcat [18] utilizes a GNN to predict whether an input contributes to concurrency coverage, while Snowplow [20] employs GNNs to predict optimal argument

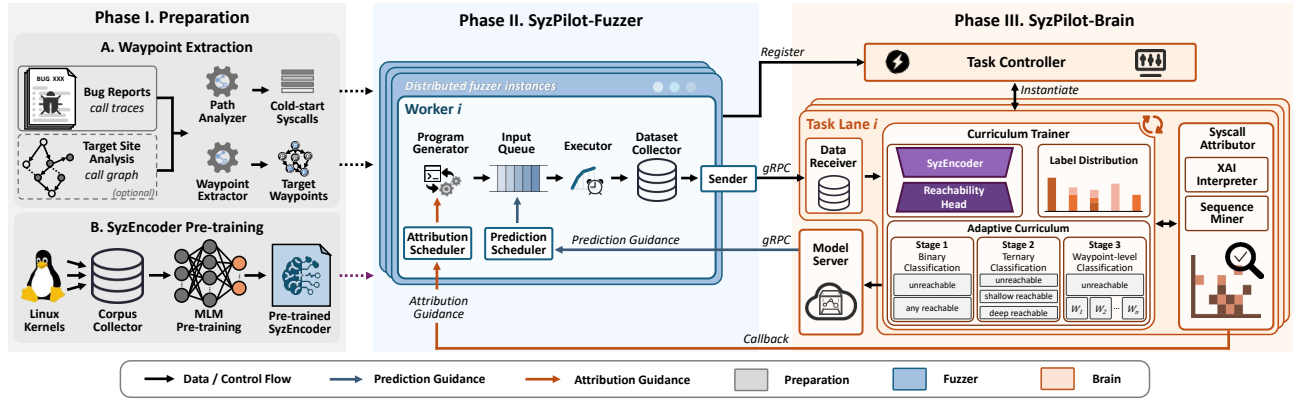


Fig. 2. System Design of SYZPILOT

mutation positions. Regarding LLM-based approaches, KernelGPT [45] leverages LLMs to infer syscall specifications, SyzGPT [46] utilizes retrieval-augmented generation (RAG) to learn syscall dependencies for generating high-quality seeds, and SyzAgent [47] uses LLMs to update choice tables by analyzing syscall relationships.

None of these learning-based frameworks are designed for directed fuzzing. They optimize for general code coverage or specification generation rather than steering execution to a specific site. Furthermore, models like Snowcat rely on offline training with fixed datasets, which limits their reuse across bug-reproduction campaigns with different target sites, kernel versions, or configurations. In contrast, SYZPILOT learns target-specific progress online and uses predictor attribution to guide mutation and increase opportunities for bug exposure.

III. DESIGN

In this section, we present the design of SYZPILOT, a target-adaptive directed kernel fuzzing framework based on reachability prediction and attribution-guided scheduling. We first provide an architectural overview (§III-A), followed by the details of our core components: waypoint extraction (§III-B), curriculum-based reachability prediction (§III-C), and attribution-guided scheduling (§III-D).

A. Overview

The system design of SYZPILOT is illustrated in Figure 2. It contains three phases: *Preparation*, *SYZPILOT-Fuzzer*, and *SYZPILOT-Brain*. In Phase I, SYZPILOT prepares target-specific guidance before fuzzing starts. Given bug reports with call traces, the Path Analyzer infers cold-start syscalls that are likely to interact with the target subsystem, and the Waypoint Extractor distills target waypoints that serve as staged progress labels. When no report is available, SYZPILOT can optionally use target-site analysis over a call graph to derive fallback waypoints (not used in the main 70-bug benchmark). In parallel, SyzEncoder is pre-trained once on syscall corpora collected from various Linux LTS versions through Masked Language Modeling (MLM), producing a reusable encoder for later target-specific reachability heads.

In Phase II, distributed SYZPILOT-Fuzzer instances execute the fuzzing loop. Each worker registers with the Task Controller, generates candidate *syz-programs*, stores them in the Input Queue, and executes selected programs in the kernel. The Dataset Collector labels each executed program by the deepest reached target waypoint and streams the labeled data to the Brain through the Sender. The fuzzer also receives two feedback channels. Prediction Guidance from the Model Server drives the Prediction Scheduler, which biases execution toward programs predicted to reach deeper waypoints. Attribution Guidance from the Brain drives the Attribution Scheduler, which adjusts syscall choice and mutation templates toward target-relevant syscalls and execution properties.

In Phase III, SYZPILOT-Brain manages one Task Lane for each registered fuzzing task. The Data Receiver stores streamed program-label pairs and reports the label distribution to the Curriculum Trainer. The trainer attaches a task-specific Reachability Head to the pre-trained SyzEncoder and adapts it through three curriculum stages: binary classification, ternary classification, and waypoint-level classification. The resulting model is deployed through the Model Server for online inference. Once enough reliable positive samples are available, the Syscall Attributor, XAI Interpreter, and Sequence Miner analyze these samples and send Attribution Guidance back to the fuzzer. This design keeps heavy learning and interpretation on the Brain while allowing the Fuzzer to continue executing programs with lightweight feedback.

Together, these phases form a closed feedback loop. Waypoint coverage provides staged progress labels despite sparse final-target hits. The predictor learns from these labels, and its predictions guide execution and mutation scheduling. Attribution and sequence mining update syscall, property, and template guidance from programs reaching intermediate waypoints or the final target. New executions then refresh the training data while random exploration remains active.

B. Preparation

We introduce *waypoints* to provide intermediate supervision for directed kernel fuzzing. Formally, given a target location (e.g., the basic block that contains the vulnerable line or

the nearest valid call-site basic block), a waypoint chain $W = \langle w_1, w_2, \dots, w_t \rangle$ is an ordered set of intermediate basic blocks, where w_t corresponds to the final target and earlier waypoints represent progressive milestone call sites. We define a waypoint at the granularity of a *call-site basic block* rather than the callee function entry. This choice preserves caller-side context and reduces label ambiguity, because the same callee may be reached from many different kernel paths. During fuzzing, each executed or predicted program is represented by the *deepest reached waypoint* in W . This progress label gives the Prediction Scheduler a staged signal: inputs that move deeper along the waypoint chain receive more execution or mutation, while inputs that remain off-route can be deprioritized without treating target reachability as equivalent to bug reproduction. Unlike function-level distance, which summarizes target proximity as a scalar for power scheduling, the waypoint chain preserves caller-sensitive milestones as categorical supervision for online prediction. Its deepest-reached index guides both execution and mutation scheduling.

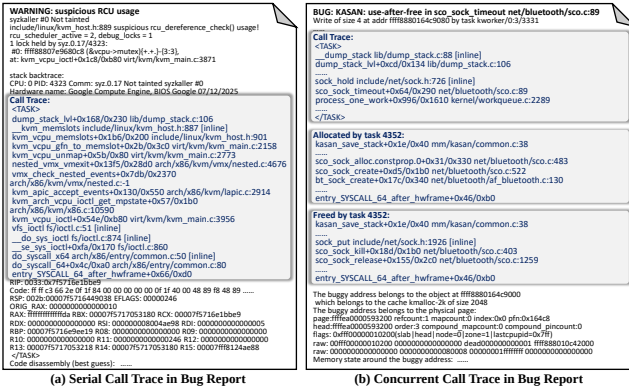


Fig. 3. Two types of bug reports: Serial and Concurrent.

Waypoint Extractor. The Waypoint Extractor derives Target Waypoints from Syzkaller-style bug reports. These reports provide high-fidelity call traces for the observed failure context, which is especially useful for bug reproduction and patch testing. SYZPILOT does not assume that reaching these waypoints is sufficient to reproduce the bug. Instead, it uses them as staged progress labels for online learning. The extractor applies five rules:

① **Call Trace Extraction.** First, we isolate the relevant call stacks from the bug report. As shown in Figure 3, reports usually follow two formats. For *serial reports* such as WARNING, BUG, GPF, or deadlock reports, the report contains a single stack delimited by keywords such as *Call Trace*, which we extract directly. For *concurrent reports* such as UAF reports, the report may contain multiple role-specific stacks, e.g., “*Allocated by*”, “*Freed by*”, and the crash stack. Since these stacks may come from different thread contexts, they should not be interpreted as one linear execution path. Instead, SYZPILOT synthesizes a *composite trace* by ordering them according to the causal structure exposed by the report, typically allocation, deallocation, and crash.

The composite trace is used as a waypoint curriculum rather than a control-flow path. It covers the major causal phases of a concurrent vulnerability, while still allowing actual executions to interleave these phases differently. During fuzzing, the reachability predictor uses the *deepest reached waypoint* as the progress label, so reaching a later-phase waypoint indicates more progress than reaching only an earlier setup-phase waypoint.

② **Trace Sanitization.** Next, we sanitize the raw trace to isolate the semantically relevant path from the syscall entry to the target site. We prune irrelevant frames in three categories: (i) *Reporting Artifacts*: Frames related to crash handling, panic sequences, and sanitizer reporting (e.g., *kasan_report*, *dump_stack*), which are post-crash side effects. (ii) *Entry Boilerplate*: Generic low-level entry points and dispatchers (e.g., *entry_SYSCALL_64*, *do_syscall_64*, and architecture-specific wrappers under *arch/*) that provide no specific guidance. (iii) *Generic Utilities*: High fan-in kernel library functions (e.g., *printk*, *kmallocc*, *kfree*), which are ubiquitous across the kernel and lack the discriminative power required for directed navigation.

③ **Complexity-based Filtering.** We filter out waypoint candidates located in functions with fewer than $B_{\min}$ basic blocks (e.g., 5). Such functions typically serve as thin wrappers, where reaching the function entry is often close to reaching its internal call site. To reduce redundancy, we prune these wrapper candidates while keeping the chain length above a minimum budget $K_{\min}$.

④ **Trivial Stage Merging.** Adjacent waypoints should represent meaningful progress rather than redundant labels for nearly identical execution states. Since runtime transition probabilities are unavailable before fuzzing, we use the *relative instruction offset* $r = \text{offset}/\text{size}$ as a lightweight redundancy signal. A low r suggests that the call site is close to the function entry, but this alone does not imply an easy transition, since an early call may still be guarded by hard-to-satisfy branches.

Therefore, SYZPILOT merges a waypoint with its predecessor only when two conditions hold: (1) the call site is near the function entry ($r < \tau_{\text{in}}$, e.g., 0.1), and (2) the intra-procedural prefix from the function entry to the call site contains no conditional branch. This straight-line-prefix check removes near-mandatory transitions while preserving early call sites that encode meaningful branching decisions. Merging is further disabled when it would reduce the chain below $K_{\min}$.

⑤ **Greedy Pruning.** Finally, we enforce a supervision budget bounded by $[K_{\min}, K_{\max}]$ (e.g., 4 to 10). If previous filtering steps reduce the chain below $K_{\min}$, the extractor returns the remaining chain. If the chain exceeds $K_{\max}$, it iteratively removes the least informative candidates based on a composite score:

$$S(w) = S_{\text{bb}}(w) - S_{\text{src}}(w) - S_{\text{triv}}(w) - S_{\text{inl}}(w)$$

This metric rewards *intra-procedural progress* (S_{bb} : basic block depth) while penalizing *source redundancy* (S_{src} : concentration in the same source file), *triviality* (S_{triv} : early-call

sites), and *compiler artifacts* (S_{ini} : inlined functions). The sub-scores use logarithmic scaling or Boolean indicators. We prune the lowest-scoring candidate until the chain satisfies K_{max} . As the last part of this stage, a Robust Z-score filter removes outlier candidates from the retained chain.

Path Analyzer for Cold-start Syscalls Inference. Although waypoints provide staged guidance, a vanilla fuzzer may still struggle to reach the first nontrivial waypoint without knowing which syscalls interact with the target subsystem. To accelerate early exploration, SYZPILOT uses the Path Analyzer to infer *Cold-start Syscalls* from the same report context used by the Waypoint Extractor, and temporarily boosts their mutation priority in the choice table.

Specifically, the Path Analyzer collects syscall entry frames and user-triggered kernel entry points from the sanitized trace, then expands the resulting set with syscall families that access the same subsystem as the selected waypoints. This analysis is intentionally lightweight: it only provides an initial generation bias, rather than precise reachability guidance. The inferred syscall set does not replace runtime coverage feedback or model-based guidance, and its effect diminishes once the Prediction Scheduler and Attribution Scheduler receive online feedback from the Brain. For continued mutation and splicing, the fuzzer can retain executed waypoint-reaching programs in a directed runtime corpus queue.

Optional Target Site Analysis. For no-report settings, such as patch testing or directed bug discovery with only a target function or source location, SYZPILOT can fall back to Target Site Analysis. Target-only mode takes an externally selected `file:line` and a call graph built from the matching kernel LLVM IR. It does not use a bug report, report call trace, prior reproducer, or PoC to construct waypoints. This module derives candidate waypoints from a kernel call graph using a path-supported backbone instead of traditional pre-dominator analysis.

We avoid pre-dominators because the Linux kernel often has many syscall entries reaching the same subsystem through different paths. In this setting, dominator relations may collapse to uninformative nodes, such as a virtual dispatch root and the target itself. Instead, SYZPILOT performs a bounded reverse traversal from the target over the kernel call graph, which can be built using KallGraph [48], and identifies wrapper-backed syscall dispatch entries. For the top entries, it samples K-shortest entry-to-target paths and normalizes their weights per entry so that each entry contributes equally. After filtering the disturbing nodes (such as compiler-runtime and generic waitqueue), it selects an ordered backbone using path support and normalized depth. If too few candidates satisfy the support and spacing constraints, the analysis returns a shorter chain rather than relaxing the thresholds. Path support is a statistic over sampled static paths, not a runtime reachability probability or dominance relation.

After assembling in a generalized progressive manner, these graph-derived waypoints can serve like the report-derived waypoints and be consumed by the subsequent pipelines of SYZPILOT. However, because they are not extracted from

a failing execution, they provide weaker evidence about the exact bug-triggering context. We therefore treat Target Site Analysis as an optional fallback for target reaching and exploratory directed testing, not as evidence that reaching a target is equivalent to reproducing a bug.

SyzEncoder Pre-training. In parallel with target preparation, SYZPILOT pre-trains SyzEncoder as a reusable representation model for kernel inputs. The Corpus Collector gathers syz-program corpora from fuzzing campaigns across multiple Linux kernel versions. To handle the large vocabulary of syscalls and arguments, we train a specialized SyzTokenizer using Byte-Pair Encoding (BPE), which efficiently segments syscall names and arguments into subword tokens. To imbue the encoder with kernel semantics, we construct a large-scale dataset of $\langle \text{program}, \text{coverage} \rangle$ by aggregating execution traces from extensive fuzzing across multiple Linux kernel versions. We pre-train SyzEncoder on this dataset using a Masked Language Modeling (MLM) objective. This enables the model to learn the latent grammar of syscalls in a self-supervised manner, yielding robust embeddings that generalize across kernel versions.

C. Curriculum-based Reachability Prediction

SYZPILOT uses an online reachability predictor to estimate whether a candidate *syz-program* is likely to make progress toward the target waypoints. The predictor is trained and served by SYZPILOT-Brain, while SYZPILOT-Fuzzer remains responsible for program generation and execution. This separation creates a feedback loop: the Dataset Collector labels executed programs by their deepest reached waypoint, the Data Receiver streams these labels to the Curriculum Trainer, the Model Server deploys the updated predictor, and the Prediction Scheduler uses the returned probabilities to guide execution and mutation.

1) *Task-specific Reachability Head:* Unlike unstructured byte inputs in user-space fuzzing, kernel inputs are structured syscall sequences with typed arguments, resource dependencies, and execution properties. Traditional byte-oriented or coverage-oriented models are poorly matched to target-specific reachability prediction because they do not directly represent progress toward a changing set of waypoints. SYZPILOT therefore builds each predictor on top of the pre-trained SyzEncoder introduced in §III-B.

For each registered task, the Curriculum Trainer attaches a lightweight SyzEncoder Reachability Head to the pre-trained encoder. The encoder processes a variable-length syscall sequence and aggregates the hidden states into a fixed-size embedding vector (e.g., dimension $d = 768$). The task-specific head then projects this vector into a probability distribution over waypoint classes. We use a linear classification head rather than recurrent or autoregressive components to keep online fine-tuning and inference practical for the Model Server.

2) *Runtime Label Generation:* During fuzzing, the Dataset Collector labels executed programs for online fine-tuning. We map each waypoint to its corresponding KCOV program-counter addresses. The collector scans each coverage trace

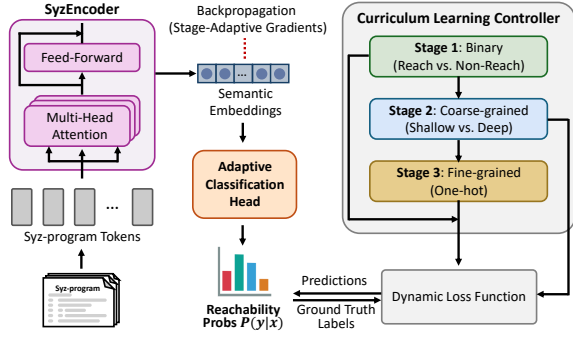


Fig. 4. The architecture of the SYZPILOT Reachability Predictor. SyzEncoder extracts semantic embeddings from syz-programs, which are then processed by an adaptive classification head. The training loss is dynamically adjusted across three curriculum stages to handle data imbalance.

and records the deepest waypoint w_i reached by the input. The input is labeled as w_i , or as *Unreachable* if no waypoint is covered. The Sender streams these program-label pairs to the task-specific Data Receiver through gRPC, and the Data Receiver reports the current label distribution to the Curriculum Trainer.

3) *Adaptive Curriculum Training*: Directly training a waypoint-level classifier from the beginning is unstable because deep waypoints are rarely visited during cold start. Prior learning-based pruning work such as FuzzGuard [17] uses static dominator structures to stage learning in user-space programs. In the kernel setting, however, multiple syscall entries, indirect calls, and version-specific build artifacts make these structures unreliable. SYZPILOT therefore uses an *Adaptive Curriculum* that increases label granularity as the online data becomes more informative.

We divide online training into the three stages shown in Figure 4. Within the curriculum stages, each waypoint class can be regarded as a progressive target. The model architecture remains fixed, and the Curriculum Trainer changes only how output logits are grouped for the loss $\mathcal{L}$.

- *Stage 1: Binary Classification*. The objective is to distinguish *Unreachable* inputs from inputs that reach any waypoint. We aggregate the logits of all waypoint classes into a single *Reachable* super-class and train a binary classifier. This stage learns a coarse reachability boundary and helps the fuzzer reduce inputs that do not enter the target subsystem.

- *Stage 2: Ternary Classification*. We partition the waypoint chain into two logical groups: *Shallow* ($\{w_1, \dots, w_{\frac{t}{2}}\}$) and *Deep* ($\{w_{\frac{t}{2}+1}, \dots, w_t\}$). The model performs a three-way classification: *Unreachable*, *Shallow reachable*, and *Deep reachable*. This stage gives the Prediction Scheduler a depth-aware signal before each individual waypoint has enough samples.

- *Stage 3: Waypoint-level Classification*. Once sufficient positive samples are available across the waypoint chain, the trainer switches to standard multi-class cross-entropy loss. Each waypoint is treated as a distinct class, enabling finer steering toward the final target w_t .

The Brain advances between stages according to the label distribution. A task moves to the next stage only when the sample counts for the relevant positive classes exceed an adaptive threshold θ , which prevents premature fine-grained training on nearly empty labels.

Handling Data Imbalance. A critical challenge in online directed fuzzing is the extreme class imbalance: unreachable inputs (negative samples) vastly outnumber reachable ones. To mitigate this, we introduce a *Self-Balancing Sampling Queue* on the SYZPILOT-Fuzzer side. This mechanism buffers execution data and dynamically throttles the transmission of negative samples based on the current abundance of positive signals, ensuring the training batch maintains a minimum positive ratio (e.g., 10%).

Additionally, the Brain uses *class-aware replay* for rare positive samples. During fine-tuning, positive examples for underrepresented waypoint classes are sampled more frequently, and the trainer can optionally apply lightweight token masking to non-structural argument subtokens, such as constants and scalar literals, while preserving syscall names, resource identifiers, and execution-property tokens. Online programs are deduplicated by content signature, retaining the deepest observed label for each program. Subsequent training iterations combine newly collected data with a bounded history of earlier samples, helping the predictor retain previously learned reachability patterns.

4) *Prediction-guided Scheduling*: The Prediction Scheduler integrates model outputs through a probabilistic scheduling policy governed by confidence scaling and temporal decay. Upon receiving Prediction Guidance from the Model Server, SYZPILOT calculates two dynamic probabilities: P_{skip} for filtering inputs predicted to be unreachable, and P_{smash} for assigning extra mutation energy to promising inputs. Both use a shared confidence scaler $\mathcal{S}(p) = 1 - (1 - p)^\beta$ and a time decay factor $\mathcal{D}(t) = e^{-\lambda t}$.

Probabilistic Pruning. For inputs predicted as *Unreachable* with probability p_θ , we compute a rejection probability:

$$P_{\text{skip}} = \alpha \cdot \mathcal{S}(p_\theta) \cdot \mathcal{D}(t).$$

The fuzzer skips the input stochastically if $\text{Rand}(0, 1) < P_{\text{skip}}$. This soft gate retains a chance to execute inputs predicted to be off-route for exploration and future training.

Depth-Weighted Boosting. For inputs predicted to be reachable, we calculate a boosting probability P_{smash} to trigger an intensive “Smash” mutation mode (similar to AFL’s havoc). To prioritize deeper progress, we aggregate waypoint probabilities weighted by their chain depth:

$$P_{\text{smash}} = \frac{\alpha}{N} \sum_{k=1}^N k \cdot \mathcal{S}(p_{w_k}) \cdot \mathcal{D}(t).$$

Inputs predicted to reach deeper waypoints therefore receive more mutation energy.

Annealing-based Exploration. Both probabilities are modulated by $\mathcal{D}(t)$. Early in the campaign, SYZPILOT uses the model more aggressively to accelerate progress. As fuzzing

progresses, the decay term reduces model intervention so that the fuzzer continues to explore paths that may be missed by an imperfect predictor.

D. Attribution-Guided Scheduling

As discussed in §I, reachability is a necessary but insufficient condition for bug reproduction. A black-box predictor can filter inputs predicted to miss the target but fails to explain why an input is reachable or how to trigger the underlying vulnerability (e.g., requiring specific arguments or concurrency). To bridge this gap, SYZPILOT employs a dual-pronged strategy, deriving mutation guidance by interpreting the gradient-based model and mining recurring patterns in the target-reaching inputs. For attribution, we likewise regard programs reaching intermediate waypoints or the final target w_t as target-reaching inputs.

1) *Interpretability-based Attribution*: Since SyzEncoder is a differentiable neural network, we can leverage gradient back-propagation to quantify the contribution of each positive input token to the model’s prediction. We design a five-stage pipeline to transform opaque gradients into actionable mutation guidance:

① *Validity Gating*. Attribution derived from an unconverged model may be unreliable. We therefore continuously monitor the predictor’s accuracy and weighted F_1 -score on a validation set throughout the curriculum learning. The attribution pipeline activates only when the model accuracy exceeds a high-confidence threshold (e.g., 85%), reducing the risk of interpreting an insufficiently trained predictor.

② *Gradient Integration*. We employ Integrated Gradients (IG) [49], an axiomatic attribution method that satisfies sensitivity and implementation invariance. Unlike simple gradient maps, IG avoids the saturation problem by accumulating gradients along a linear path from a baseline x' (zero-padding) to the input x . Specifically, we compute *Layer Integrated Gradients* at the embedding level:

$$\text{Attr}(x_i) = (x_i - x'_i) \times \int_{\alpha=0}^1 \frac{\partial F(x' + \alpha \times (x - x'))}{\partial x_i} d\alpha$$

where F is the predicted probability for the target waypoint. We approximate this integral via a Riemann sum with m (e.g., 50) equidistant steps, producing a detailed attribution tensor of shape (L, D) , where L is sequence length and D is the embedding dimension.

③ *Token-to-Syscall Projection*. The raw gradient tensor represents sub-token importance. To derive semantic meaning, we first sum the tensor along the embedding dimension to obtain scalar token scores. Since our SyzTokenizer segments complex syscalls into multiple sub-words (e.g., *ioctl*, *KVM*, *IRQ*), we aggregate the scores of all constituent tokens belonging to the same syscall invocation to derive a holistic *syscall-level importance score* for guidance.

④ *Property Attribution*. In addition to syscall itself, we explicitly parse dedicated tokens representing control-flow call properties, which consist of *async* (threading), *fail_nth*

(fault injection), and *rerun* (looping). By accumulating gradients on these specific tokens, we estimate each property’s contribution to the reachability prediction. For instance, a high attribution score on *async* prompts the scheduler to prioritize concurrent execution for the corresponding syscall. Although the score guides mutation, it does not establish that concurrency is necessary for bug exposure.

⑤ *Batch-Level Consensus*. Finally, to mitigate the noise of individual samples, we aggregate these scores across all true-positive inputs collected in the current batch. We apply L_2 normalization to generate a global ranking of syscalls and properties by their aggregated attribution scores, which is then synchronized with the fuzzer. Figure 5 shows a representative example of token-level attribution for a syz-program. Although IG also assigns scores to argument tokens, SYZPILOT currently does not translate them into field-level payload mutations or explicitly model object and resource dependencies.

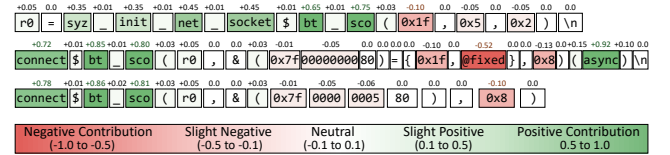


Fig. 5. Token-level attribution for a syz-program.

2) *Statistics-based Attribution*: Gradient attribution captures nonlinear relationships learned by the predictor but can be sensitive to local perturbations. We therefore complement it with statistical pattern mining to extract recurring patterns from the reachable syz-programs.

Frequency-based Syscall Ranking. First, we perform frequency analysis on the filtered positive corpus to identify “hot” syscalls. They typically represent setup primitives (e.g., resource handle creation) that are used for interacting with the target subsystem.

Gap-Constrained Pattern Mining. Rigid substring matching fails to capture kernel dependencies, which are often interleaved with irrelevant noise calls. We formulate this task as *Gap-Constrained Sequential Pattern Mining* [50]. Let $S = [s_1, s_2, \dots, s_n]$ be a trace and $P = [p_1, p_2, \dots, p_m]$ be a candidate pattern. S supports P if there exists an index sequence $i_1 < i_2 < \dots < i_m$ such that $s_{i_j} = p_j$ and the gap constraint holds:

$$i_{j+1} - i_j - 1 \leq \delta_{\max}$$

where $\delta_{\max}$ defines the tolerance for intervening noise. We implement a specialized mining algorithm based on *pattern-growth* and Breadth-First Search (BFS). It iteratively extends candidate patterns and prunes those falling below an adaptive support threshold. The output is a set of distinct skeleton sequences that frequently appear in target-reaching programs.

3) *Scheduling Integration with Fuzzer*: The Brain periodically synthesizes insights from both gradient and statistical channels, pushing a synchronized guidance strategy to the Fuzzer. SYZPILOT integrates these attribution results into the mutation engine via three active mechanisms:

Attribution-Weighted Mutation. Instead of uniform random selection, the fuzzer prioritizes mutations on high-attribution syscalls. We construct a weighted distribution map where syscalls with high IG scores (e.g., specific `ioctl` commands) are assigned higher selection probabilities during insertion and splicing operations without excluding random exploration.

Property-Driven Mutation. If the attribution analysis indicates a high importance for `async`, `fail_nth`, or `rerun`, the fuzzer activates a *Forced Property Mode* and prioritizes threading wrappers, fault injection, or looping descriptors for the corresponding high-attribution syscalls.

Skeleton Injection. The statistical patterns serve as mutation templates. During generation, the fuzzer probabilistically splices these full patterns into current seeds, preserving the frequent syscall orderings (e.g., `open` $\rightarrow$ $\dots$ $\rightarrow$ `mmap` $\rightarrow$ $\dots$ $\rightarrow$ `write`) that random single-point mutations may disrupt.

By closing the loop with attribution, SYZPILOT transforms the predictor from a passive filter into an active guide, explicitly steering the fuzzer to bias mutation toward execution contexts associated with the targets.

IV. IMPLEMENTATION

We implemented SYZPILOT-Fuzzer by extending Syzkaller (6e83b42) and implemented SYZPILOT-Brain as a Python-based backend for training, attribution, and model serving. The system contains approximately 10.6K lines of Go and Python code.

Waypoint Extraction. The Waypoint Extractor is a standalone Python module that uses targeted disassembly rather than full-binary analysis. It uses `nm` to locate function boundaries and invokes `objdump` and `addr2line` only on relevant address ranges. It then applies the extraction and pruning rules in §III-B, together with a Robust Z-score filter [51], to remove outlier waypoint candidates. With caching and parallel processing, the end-to-end extraction time is about one minute per target.

Reachability Predictor. The neural components are implemented with PyTorch and the Transformers library. We train SyzTokenizer with Byte-Pair Encoding using the *tokenizers* library. SyzEncoder is pre-trained from StarEncoder [52] with Masked Language Modeling on 2.4 million syz-programs collected from three Linux LTS kernels (5.15, 6.6, and 6.12). The encoder-only architecture supports our sequence classification task [53], and its base parameter scale (125M) permits practical online updates. Syz is not among the programming languages listed in the StarEncoder model card [54]. We train SyzTokenizer and pre-train SyzEncoder once, then reuse them to initialize each target-specific predictor. For each target, SYZPILOT attaches a lightweight reachability head and flexibly fine-tunes this head and the last two SyzEncoder layers online for efficiency, with a learning rate of $1e^{-5}$ and a batch size of 128. The linear head retains one output per waypoint plus one for *Unreachable* across all stages, grouping logits for the binary and ternary losses so that later stages can continue from the preceding stage’s accepted checkpoint.

Syscall Attributor. We implement gradient attribution with *Captum* by adapting `LayerIntegratedGradients` to the embedding layer of SyzEncoder. The attributor processes padded batches of variable-length syz-programs and approximates Integrated Gradients with 50 steps. For statistics-based attribution, we implement a gap-constrained Longest Common Subsequence algorithm in Python to mine frequent syscall patterns from target-reaching samples. We set the maximum pattern length to 10 in our experiments.

Brain Controller and Fuzzer Integration. SYZPILOT-Brain is orchestrated by a *FastAPI* controller that handles task registration, receiver creation, port management, training launch, attribution, and model deployment. The fuzzer streams labeled training data to the Brain through *gRPC*, and the trained predictor is served through *TorchServe* [55] for online inference, with support for hot model updates. On the fuzzer side, we add *syzpilotlib* to Syzkaller and modify the coverage collection, input queue, filtering, and mutation modules. We also implement the directed corpus as a bounded and runtime-only addition to Syzkaller’s native corpus, with mutation seeds selected probabilistically from either corpus. The Brain starts online training when the unique-program count and active-class support meet their thresholds, or when a 30-minute warm-up fallback timer expires. We use fixed scheduling parameters for all targets, with $\alpha = 0.6$, $\beta = 0.6$, and $\lambda = 0.03$ in §III-C4. Here, α bounds the strength of model intervention, β controls confidence scaling, and λ controls hourly annealing. We do not tune these values per bug, subsystem, kernel version, or binary. The waypoint budgets, curriculum threshold, and Integrated Gradients steps are likewise fixed across the benchmark. None of the 70 benchmark targets or their kernel binaries was used to design, develop, or tune SYZPILOT.

V. EVALUATION

In this section, we evaluate SYZPILOT to answer the following key research questions:

- **RQ1: Target Reachability.** How effectively does SYZPILOT reach target locations? §V-B
- **RQ2: Bug Reproduction.** How effectively does SYZPILOT reproduce target bugs? §V-C
- **RQ3: Ablation Study.** How does each component contribute to SYZPILOT’s performance? §V-D
- **RQ4: Performance Overhead.** What cost does SYZPILOT add? §V-E

A. Experimental Setup

Evaluation Baselines. We constructed a benchmark of 70 real-world kernel vulnerabilities that affect different kernel versions (i.e., v5.10-v6.14) to evaluate target reachability and bug reproduction. This dataset comprises 60 bugs randomly selected from the SyzDirect dataset [56] and 10 vulnerabilities (from ID 61 to 70) reported by our historical fuzzing campaigns. These additional cases extend kernel-version coverage to newer releases. Benchmark details are available in our artifact. We compare SYZPILOT with Syzkaller, SyzDirect, and Snowplow [20]. Syzkaller is a continuously maintained

coverage-guided kernel fuzzer and is the default basis for syzbot. SyzDirect is a representative directed kernel fuzzer that uses static analysis and distance-based scheduling. Snowplow is a learning-assisted kernel fuzzer that predicts argument mutation positions. We exclude SyzLego [34] because its source code is not publicly available, and DRLF-C [37] because it targets the RT-Thread embedded kernel rather than Linux.

Evaluation Metrics. We use the following metrics [57] to measure target reachability and bug reproduction.

- *Average Number of Hit Executions ($\overline{N}_h$):* The average number of program executions whose KCOV traces cover the target basic block within the configured time budget. A larger $\overline{N}_h$ means that the fuzzer exercises the target site more frequently.
- *Average Time-to-Hit (μTTH):* The average wall-clock time required to hit the target basic block across repeated runs. Timed-out runs receive the configured time limit as the penalty value, which is 24 hours for the main runs and 48 hours for the compensated baseline runs. Lower μTTH means faster first-hit target reaching.
- *Average Time-to-Exposure (μTTE):* The average time required to trigger the target bug report across repeated runs. As with μTTH , timed-out runs receive the configured time-limit penalty. Lower μTTE means faster bug reproduction.
- *Target- and Run-Level Success Probabilities:* P_t^h and P_t^e denote the fractions of the targets hit and exposed in at least one run, while P_r^h and P_r^e denote the corresponding success fractions across total runs.
- *Speedup and Relative Improvement:* Time-to-X speedups are geometric means¹ of target-level mean-time ratios between each baseline and SYZPILOT. For $\overline{N}_h$, we use target-equal geometric relative improvement on the $\log(1 + \overline{N}_h)$ scale.
- *Statistical Comparison:* We adopt hierarchical-bootstrap 95% confidence intervals, target-equal Vargha–Delaney A_{12} effect sizes, and Holm-corrected p-values from target-stratified permutation tests to justify the evaluation [57].

Environment and Configuration. We deploy SYZPILOT in a distributed setting. All fuzzing instances (SYZPILOT-Fuzzer) run on an Ubuntu 22.04 server with an Intel Xeon Platinum 8268 CPU (192 cores) and 320GB RAM. SYZPILOT-Brain, which handles training, inference, and attribution, runs on a separate node with an Intel Xeon Platinum 8280 CPU (112 cores), 512GB RAM, and two NVIDIA A800 (80GB) GPUs.

Each fuzzer uses one QEMU virtual machine with 2 vCPUs, 4GB RAM, and 8 Syzkaller procs. We disable Syzkaller’s reproduce mode for all evaluated fuzzers because it spends additional resources trying to generate PoCs for untargeted crashes. To avoid ambiguity, we use “bug reproduction” to mean reporting the target bug during fuzzing, not generating a stable PoC. We run SYZPILOT for 24 hours per target and repeat each experiment ten times. Snowplow uses the same fuzzing and serving configuration as SYZPILOT and is evaluated under the same 24-hour budget. For Syzkaller and SyzDirect, we report both 24-hour runs and 48-hour

compensated runs. The 24-hour runs are the main comparison, while the 48-hour runs test whether the CPU-only baselines can close the gap with additional fuzzing time to compensate for the GPU-side SYZPILOT-Brain. All fuzzers start from the same default Syzkaller seeds, and all per-target averages are computed over ten repeated runs.

B. Target Reachability

Reaching the target basic block is a prerequisite for bug reproduction, but it is not equivalent to reproducing the bug. We evaluate target reachability on the 70-bug benchmark using average time-to-hit (μTTH), successful hit runs, and the average number of hit executions ($\overline{N}_h$). For each report-based target, we use the last valid call-site basic block near the reported failure site. When the original site is inlined or unavailable to instrumentation, we use the nearest non-inlined caller. The same target location is used for all fuzzers.

Because this benchmark is report-based, SYZPILOT uses report-derived waypoints in this experiment. This 70-bug benchmark therefore evaluates report-driven mode. Appendix A-D separately compares report-derived and target-only modes on the five targets used in the ablation study. SYZPILOT and our patched Syzkaller record TTH and $\overline{N}_h$ from KCOV traces. All fuzzers are limited to the same number of CPU cores by `taskset`, and the 48-hour baseline runs are reported separately as compensation experiments.

Table II reports μTTH and $\overline{N}_h$ for the 59 targets reached by at least one fuzzer configuration. The other 11 targets are omitted because no configuration reaches them. Under the 24-hour budget, SYZPILOT reaches 57 of 70 targets in 515 of 700 runs, with $P_t^h=81.43\%$ and $P_r^h=73.57\%$. Snowplow, SyzDirect, and Syzkaller reach 36 (51.43%), 44 (62.86%), and 43 (61.43%) targets in 245 (35.00%), 382 (54.57%), and 308 (44.00%) runs, respectively.

Table III reports the pairwise 24-hour comparisons. On the $\log_{1p}$ scale, SYZPILOT improves $\overline{N}_h$ by 261.26 $\times$ over Snowplow, 6.87 $\times$ over SyzDirect, and 139.76 $\times$ over Syzkaller, with A_{12} values from 0.79 to 0.91. Its first-hit results are mixed. SYZPILOT is 4.20 $\times$ and 4.23 $\times$ faster than Snowplow and Syzkaller, while the 0.76 $\times$ point estimate against SyzDirect does not indicate a TTH improvement and its 95% confidence interval includes 1.

The per-target results distinguish first-hit speed from repeated target exercise. SyzDirect can reach shallow targets quickly through static syscall-entry and distance guidance, whereas SYZPILOT refines its initial signal after collecting runtime feedback. This explains why SYZPILOT is not uniformly faster on first hit but produces more target-reaching executions than each baseline. A single hit establishes reachability, while repeated hits provide more opportunities to explore the execution context needed for exposure. In the 48-hour compensation runs, SyzDirect reaches 44 targets in 398 runs and Syzkaller reaches 46 targets in 361 runs, both below SYZPILOT’s 57 targets and 515 runs at 24 hours. We report these counts separately and do not compute time speedups across unequal timeout budgets.

¹The geometric mean is appropriate for multiplicative ratios because it treats a $k\times$ speedup and a $1/k$ slowdown symmetrically.

TABLE II
PERFORMANCE OF TARGET REACHABILITY ON THE 59 TARGETS REACHED BY AT LEAST ONE FUZZER.

ID	Fuzzer	Run	μ TTH (h)	N_h	ID	Fuzzer	Run	μ TTH (h)	N_h	ID	Fuzzer	Run	μ TTH (h)	N_h	ID	Fuzzer	Run	μ TTH (h)	N_h	ID	Fuzzer	Run	μ TTH (h)	N_h					
1	SP	10/10	0.09	29,450	14	SP	10/10	0.41	212,410	27	SP	0/10	24	0	38	SP	10/10	0.99	55,755	51	SP	10/10	2.69	204,518	61	SP	10/10	0.12	5,653
	SN	10/10	0.02	8,139		SN	10/10	1.05	150		SN	0/10	24	0		SN	1/10	11.85	340		SN	0/10	24	0					
	SD24h	10/10	2.58	20,958		SD24h	10/10	0.02	241,200		SD24h	0/10	24	0		SD24h	10/10	0.03	38,447		SD24h	1/10	22.92	1,292					
	SD48h	10/10	2.58	24,739		SD48h	10/10	0.02	584,653		SD48h	0/10	48	0		SD48h	10/10	0.03	56,701		SD48h	2/10	42.66	3,773					
	SK24h	10/10	0.10	3,562		SK24h	10/10	2.16	299		SK24h	5/10	12.01	2,670		SK24h	10/10	7.98	123		SK24h	0/10	24	0					
3	SP	5/10	14.89	804	17	SP	10/10	0.78	628	28	SP	10/10	0.12	234,704	39	SP	10/10	1.23	166,236	52	SP	9/10	2.80	6,845	62	SP	3/10	16.99	5,702
	SN	0/10	24	0		SN	10/10	0.02	73		SN	0/10	24	0		SN	1/10	23.98	1		SN	10/10	1.21	1,496		SN	10/10	0.02	187,234
	SD24h	0/10	24	0		SD24h	8/10	9.26	26		SD24h	10/10	0.02	221,648		SD24h	10/10	0.06	134,180		SD24h	4/10	14.41	3,273		SD24h	1/10	22.22	981
	SD48h	0/10	48	0		SD48h	10/10	9.26	36		SD48h	10/10	0.02	318,685		SD48h	10/10	0.06	260,490		SD48h	4/10	28.81	4,551		SD48h	2/10	42.27	2,916
	SK24h	0/10	24	0		SK24h	10/10	1.90	71		SK24h	2/10	21.82	487		SK24h	5/10	18.63	67		SK24h	10/10	0.86	1,924		SK24h	0/10	24	0
4	SP	10/10	0.62	855,200	18	SP	10/10	0.25	159,746	29	SP	10/10	0.01	252,337	40	SP	10/10	1.40	16,962	53	SP	10/10	0.04	1,561	63	SP	10/10	0.11	20,991
	SN	0/10	24	0		SN	10/10	3.76	2,629		SN	10/10	0.02	72,021		SN	0/10	24	0		SN	9/10	2.58	77		SN	10/10	0.03	14,267
	SD24h	10/10	1.07	580,430		SD24h	10/10	0.14	78,300		SD24h	10/10	0.03	226,655		SD24h	9/10	2.70	12,892		SD24h	10/10	0.04	1,126		SD24h	0/10	24	0
	SD48h	10/10	1.07	743,519		SD48h	10/10	0.14	133,411		SD48h	10/10	0.03	498,902		SD48h	10/10	3.23	19,419		SD48h	10/10	0.04	1,143		SD48h	0/10	48	0
	SK24h	0/10	24	0		SK24h	10/10	2.89	1,902		SK24h	10/10	0.02	44,754		SK24h	0/10	24	0		SK24h	10/10	0.11	203		SK24h	10/10	0.12	2,463
6	SP	10/10	0.12	581,066	19	SP	7/10	11.29	746	30	SP	9/10	10.20	421,302	41	SP	10/10	6.63	2,945	54	SP	10/10	0.13	8,312	64	SP	10/10	8.01	2,114
	SN	0/10	24	0		SN	0/10	24	0		SN	0/10	24	0		SN	4/10	17.04	13		SN	10/10	0.76	1,791		SN	4/10	20.87	20
	SD24h	10/10	0.06	305,095		SD24h	0/10	24	0		SD24h	9/10	9.80	261,583		SD24h	8/10	8.72	3,537		SD24h	10/10	0.05	6,897		SD24h	0/10	24	0
	SD48h	10/10	0.06	500,570		SD48h	0/10	48	0		SD48h	10/10	0.18	412,687		SD48h	10/10	9.92	8,148		SD48h	10/10	0.05	7,157		SD48h	0/10	48	0
	SK24h	0/10	24	0		SK24h	3/10	20.48	95		SK24h	9/10	13.53	71		SK24h	4/10	20.35	2		SK24h	10/10	0.55	1,330		SK24h	10/10	0.29	952
7	SP	10/10	7.57	5,946	21	SP	10/10	0.60	290	31	SP	10/10	7.56	4,052	42	SP	10/10	0.13	144,114	55	SP	10/10	1.11	1,087	65	SP	2/10	20.88	22
	SN	0/10	24	0		SN	10/10	0.03	843		SN	1/10	23.12	5		SN	10/10	0.03	1,283		SN	1/10	22.25	29		SN	1/10	22.45	2
	SD24h	0/10	24	0		SD24h	9/10	3.78	46		SD24h	2/10	22.91	2,815		SD24h	10/10	0.02	79,303		SD24h	10/10	0.03	592		SD24h	0/10	24	0
	SD48h	0/10	48	0		SD48h	9/10	6.18	39		SD48h	2/10	42.11	10,662		SD48h	10/10	0.02	130,487		SD48h	10/10	0.03	704		SD48h	0/10	48	0
	SK24h	0/10	24	0		SK24h	8/10	4.95	184		SK24h	2/10	21.72	39		SK24h	10/10	0.58	854		SK24h	2/10	22.75	3		SK24h	1/10	22.51	15
8	SP	5/10	17.95	88	22	SP	5/10	16.47	85	32	SP	10/10	5.22	1,458	43	SP	10/10	1.43	27,474	56	SP	10/10	0.08	26,395	66	SP	7/10	8.59	884
	SN	0/10	24	0		SN	7/10	15.82	47		SN	7/10	14.53	84		SN	0/10	24	0		SN	0/10	24	0		SN	6/10	15.84	130
	SD24h	0/10	24	0		SD24h	0/10	24	0		SD24h	8/10	12.00	395		SD24h	10/10	0.03	40,046		SD24h	10/10	0.03	19,809		SD24h	0/10	24	0
	SD48h	0/10	48	0		SD48h	0/10	48	0		SD48h	9/10	14.79	571		SD48h	10/10	0.03	76,023		SD48h	10/10	0.03	25,573		SD48h	0/10	48	0
	SK24h	0/10	24	0		SK24h	5/10	17.00	41		SK24h	9/10	9.24	109		SK24h	0/10	24	0		SK24h	0/10	24	0		SK24h	8/10	13.00	56
9	SP	8/10	16.50	402	23	SP	10/10	0.49	867,322	33	SP	10/10	1.60	41,319	46	SP	10/10	0.51	11,992	57	SP	10/10	4.46	1,520	67	SP	0/10	24	0
	SN	0/10	24	0		SN	10/10	9.23	544		SN	5/10	17.87	93		SN	0/10	24	0		SN	0/10	24	0		SN	2/10	22.24	8
	SD24h	0/10	24	0		SD24h	10/10	0.08	496,438		SD24h	10/10	0.02	23,978		SD24h	10/10	0.02	13,193		SD24h	8/10	12.30	430		SD24h	0/10	24	0
	SD48h	0/10	48	0		SD48h	10/10	0.08	971,718		SD48h	10/10	0.02	47,562		SD48h	10/10	0.02	38,044		SD48h	9/10	16.50	556		SD48h	0/10	48	0
	SK24h	0/10	24	0		SK24h	10/10	10.94	184		SK24h	8/10	13.14	50		SK24h	0/10	24	0		SK24h	7/10	14.90	39		SK24h	1/10	22.29	13
11	SP	10/10	0.37	22,738	24	SP	10/10	0.62	56,507	35	SP	10/10	2.58	4,589	47	SP	5/10	15.06	220	58	SP	10/10	0.82	424,329	68	SP	1/10	22.35	4
	SN	9/10	13.93	232		SN	10/10	0.77	1,787		SN	0/10	24	0		SN	0/10	24	0		SN	0/10	24	0		SN	0/10	24	0
	SD24h	0/10	24	0		SD24h	10/10	0.08	146,718		SD24h	10/10	1.53	665		SD24h	2/10	19.75	41		SD24h	10/10	0.01	220,970		SD24h	0/10	24	0
	SD48h	0/10	48	0		SD48h	10/10	0.08	309,871		SD48h	10/10	1.53	574		SD48h	4/10	34.29	122		SD48h	10/10	0.01	471,668		SD48h	0/10	48	0
	SK24h	0/10	0.60	3,464		SK24h	10/10	0.95	2,976		SK24h	2/10	21.96	5		SK24h	0/10	24	0		SK24h	0/10	24	0		SK24h	0/10	24	0
12	SP	10/10	1.17	21,890	25	SP	10/10	0.60	7,981	36	SP	10/10	1.13	5,813	48	SP	10/10	5.48	488	59	SP	10/10	1.73	441	69	SP	9/10	3.35	1,569
	SN	10/10	6.67	1,862		SN	8/10	8.47	1,832		SN	0/10	24	0		SN	2/10	22.50	11		SN	0/10	24	0		SN	0/10	24	0
	SD24h	0/10	0.19	7,302		SD24h	10/10	0.04	5,567		SD24h	10/10	2.14	5,272		SD24h	5/10	13.05	16		SD24h	10/10	0.25	61		SD24h	0/10	24	0
	SD48h	0/10	0.19	15,762		SD48h	10/10	0.04	8,144		SD48h	10/10	2.14	6,769		SD48h	8/10	22.01	52		SD48h	10/10	0.25	90		SD48h	0/10	48	0
	SK24h	0/10	2.81	1,005		SK24h	5/10	12.32	1,166		SK24h	0/10	24	0		SK24h	8/10	14.65	117		SK24h	1/10	23.38	4		SK24h	9/10	2.65	491
13	SP	10/10	1.67	21,890	26	SP	10/10	0.60	7,981	37	SP	10/10	1.13	5,813	49	SP	10/10	5.48	488	60	SP	10/10	1.73	441	70	SP	9/10	3.35	1,569
	SN	10/10	6.67	1,862		SN	8/10	8.47	1,832		SN	0/10	24	0		SN	2/10	22.50	11		SN	0/10	24	0		SN	0/10	24	0
	SD24h	0/10	0.19	7,302		SD24h	10/10	0.04	5,567		SD24h	10/10	2.14	5,272		SD24h	5/10	13.05	16		SD24h	10/10	0.25	61		SD24h	0/10	24	0
	SD48h	0/10	0.19	15,762		SD48h	10/10	0.04	8,144		SD48h	10/10	2.14	6,769		SD48h	8/10	22.01	52		SD48h	10/10	0.25	90		SD48h	0/10	48	0
	SK24h	0/10	2.81	1,005		SK24h	5/10	12.32	1,166		SK24h	0/10	24	0		SK24h	8/10	14.65	117		SK24h	1/10	23.38	4		SK24h	9/10	2.65	491
13	SP	10/10	2.37	59,360	26	SP	10/10	0.02	352,344	37	SP	10/10	0.23	13,400	49	SP	10/10	7.51	221,065	60	SP	10/10	3.79	12,959	70	SP	10/10	3.79	12,959
	SN	10/10	5.68	726		SN	10/10	0.04	58,911		SN	1/10	23.24	13		SN	2/10	22.70	6		SN	4/10	21.32	130,512		SN	4/10	21.32	130,512
	SD24h	0/10	2.61	42,182		SD24h	10/10	0.03	250,076		SD24h	10/10	0.01	7,438		SD24h	8/10	11.95	193,173		SD24h	10/10	0.63	114,356		SD24h	10/10	0.63	114,356
	SD48h	0/10	2.61	79,560		SD48h	10/10	0.03	416,369		SD48h	10/10	0.01	14,822		SD48h	10/10	14.91	408,318		SD48h	10/10	0.63	203,776		SD48h	10/10	0.63	203,776
	SK24h	0/10	4.99	398		SK24h	10/10	0.03	62,011		SK24h	4/10	20.01	10		SK24h	6/10	17.30	41		SK24h	4/10	20.38	33		SK24h	4/10	20.	

TABLE IV
PERFORMANCE OF BUG REPRODUCTION ON THE 41 TARGETS REPRODUCED BY AT LEAST ONE FUZZER.

ID	Fuzzer	Run	μ TTE (h)	Speedup	ID	Fuzzer	Run	μ TTE (h)	Speedup	ID	Fuzzer	Run	μ TTE (h)	Speedup	ID	Fuzzer	Run	μ TTE (h)	Speedup	ID	Fuzzer	Run	μ TTE (h)	Speedup					
1	SP	10/10	10.33	—	13	SP	7/10	14.17	—	26	SP	0/10	24	—	37	SP	10/10	6.61	—	49	SP	3/10	23.16	—	61	SP	10/10	16.59	—
	SN	0/10	24	2.32		SN	0/10	24	1.69		SN	1/10	23.67	0.99		SN	0/10	24	3.63		SN	0/10	24	1.04		SN	0/10	24	1.45
	SD _{24h}	0/10	24	2.32		SD _{24h}	2/10	20.78	1.47		SD _{24h}	0/10	24	1.00		SD _{24h}	6/10	10.61	1.60		SD _{24h}	2/10	20.42	0.88		SD _{24h}	0/10	24	1.45
	SD _{48h}	0/10	48	—		SD _{48h}	6/10	34.17	—		SD _{48h}	0/10	48	—		SD _{48h}	10/10	16.56	—		SD _{48h}	4/10	38.45	—		SD _{48h}	0/10	48	—
	SK _{24h}	0/10	24	2.32		SK _{24h}	0/10	24	1.69		SK _{24h}	2/10	23.84	0.99		SK _{24h}	0/10	24	3.63		SK _{24h}	0/10	24	1.04		SK _{24h}	0/10	24	1.45
SK _{48h}	0/10	48	—	SK _{48h}	0/10	48	—	SK _{48h}	2/10	43.04	—	SK _{48h}	0/10	48	—	SK _{48h}	0/10	48	—	SK _{48h}	0/10	48	—	SK _{48h}	0/10	48	—		
3	SP	5/10	17.07	—	17	SP	7/10	9.93	—	28	SP	8/10	11.30	—	38	SP	6/10	20.08	—	51	SP	0/10	24	—	62	SP	0/10	24	—
	SN	0/10	24	1.41		SN	0/10	24	2.42		SN	0/10	24	2.12		SN	0/10	24	1.20		SN	0/10	24	1.00		SN	7/10	11.82	0.49
	SD _{24h}	0/10	24	1.41		SD _{24h}	0/10	24	2.42		SD _{24h}	1/10	22.54	2.00		SD _{24h}	0/10	24	1.20		SD _{24h}	3/10	18.66	0.78		SD _{24h}	0/10	24	1.00
	SD _{48h}	0/10	48	—		SD _{48h}	0/10	48	—		SD _{48h}	1/10	44.14	—		SD _{48h}	0/10	48	—		SD _{48h}	8/10	34.50	—		SD _{48h}	0/10	48	—
	SK _{24h}	0/10	24	1.41		SK _{24h}	0/10	24	2.42		SK _{24h}	2/10	22.82	2.02		SK _{24h}	0/10	24	1.20		SK _{24h}	0/10	24	1.00		SK _{24h}	0/10	24	1.00
SK _{48h}	0/10	48	—	SK _{48h}	0/10	48	—	SK _{48h}	2/10	27.14	—	SK _{48h}	0/10	48	—	SK _{48h}	0/10	48	—	SK _{48h}	0/10	48	—	SK _{48h}	0/10	48	—		
4	SP	4/10	18.52	—	19	SP	6/10	15.83	—	30	SP	5/10	16.91	—	39	SP	8/10	9.84	—	52	SP	0/10	24	—	63	SP	7/10	8.55	—
	SN	0/10	24	1.30		SN	0/10	24	1.52		SN	0/10	24	1.42		SN	0/10	24	2.44		SN	0/10	24	1.00		SN	0/10	24	2.81
	SD _{24h}	0/10	24	1.30		SD _{24h}	0/10	24	1.52		SD _{24h}	2/10	22.43	1.33		SD _{24h}	6/10	14.49	1.47		SD _{24h}	1/10	21.62	0.90		SD _{24h}	0/10	24	2.81
	SD _{48h}	0/10	48	—		SD _{48h}	0/10	48	—		SD _{48h}	2/10	41.63	—		SD _{48h}	10/10	21.51	—		SD _{48h}	1/10	43.22	—		SD _{48h}	0/10	48	—
	SK _{24h}	0/10	24	1.30		SK _{24h}	3/10	22.40	1.41		SK _{24h}	0/10	24	1.42		SK _{24h}	1/10	23.95	2.43		SK _{24h}	0/10	24	1.00		SK _{24h}	2/10	22.99	2.69
SK _{48h}	0/10	48	—	SK _{48h}	3/10	39.20	—	SK _{48h}	0/10	48	—	SK _{48h}	1/10	45.55	—	SK _{48h}	0/10	48	—	SK _{48h}	2/10	42.19	—	SK _{48h}	2/10	42.19	—		
6	SP	7/10	20.78	—	21	SP	10/10	0.97	—	31	SP	6/10	17.60	—	40	SP	4/10	21.84	—	55	SP	8/10	10.24	—	64	SP	6/10	16.29	—
	SN	0/10	24	1.16		SN	10/10	1.36	1.40		SN	0/10	24	1.36		SN	0/10	24	1.10		SN	1/10	22.67	2.21		SN	0/10	24	1.47
	SD _{24h}	0/10	24	1.16		SD _{24h}	8/10	4.92	5.08		SD _{24h}	0/10	24	1.36		SD _{24h}	4/10	14.56	0.67		SD _{24h}	3/10	16.87	1.65		SD _{24h}	0/10	24	1.47
	SD _{48h}	0/10	48	—		SD _{48h}	9/10	7.62	—		SD _{48h}	0/10	48	—		SD _{48h}	10/10	26.80	—		SD _{48h}	4/10	31.79	—		SD _{48h}	0/10	48	—
	SK _{24h}	0/10	24	1.16		SK _{24h}	7/10	9.70	10.01		SK _{24h}	0/10	24	1.36		SK _{24h}	0/10	24	1.10		SK _{24h}	2/10	21.31	2.08		SK _{24h}	0/10	24	1.47
SK _{48h}	0/10	48	—	SK _{48h}	7/10	16.90	—	SK _{48h}	0/10	48	—	SK _{48h}	2/10	45.62	—	SK _{48h}	5/10	36.92	—	SK _{48h}	0/10	48	—	SK _{48h}	0/10	48	—		
8	SP	4/10	20.34	—	23	SP	10/10	1.39	—	32	SP	8/10	13.42	—	41	SP	6/10	18.43	—	56	SP	8/10	6.75	—	66	SP	7/10	14.02	—
	SN	0/10	24	1.18		SN	1/10	23.67	17.01		SN	0/10	24	1.79		SN	0/10	24	1.30		SN	0/10	24	3.56		SN	0/10	24	1.71
	SD _{24h}	0/10	24	1.18		SD _{24h}	10/10	2.28	1.64		SD _{24h}	5/10	19.11	1.42		SD _{24h}	0/10	24	1.30		SD _{24h}	4/10	14.45	2.14		SD _{24h}	0/10	24	1.71
	SD _{48h}	0/10	48	—		SD _{48h}	10/10	2.28	—		SD _{48h}	8/10	28.38	—		SD _{48h}	2/10	44.07	—		SD _{48h}	4/10	28.85	—		SD _{48h}	0/10	48	—
	SK _{24h}	0/10	24	1.18		SK _{24h}	0/10	24	17.24		SK _{24h}	0/10	24	1.79		SK _{24h}	1/10	23.67	1.28		SK _{24h}	0/10	24	3.56		SK _{24h}	0/10	24	1.71
SK _{48h}	0/10	48	—	SK _{48h}	0/10	48	—	SK _{48h}	0/10	48	—	SK _{48h}	1/10	45.27	—	SK _{48h}	0/10	48	—	SK _{48h}	0/10	48	—	SK _{48h}	0/10	48	—		
11	SP	3/10	22.60	—	24	SP	9/10	6.12	—	35	SP	6/10	14.20	—	47	SP	5/10	15.61	—	59	SP	0/10	24	—	69	SP	7/10	13.07	—
	SN	0/10	24	1.06		SN	0/10	24	3.92		SN	0/10	24	1.69		SN	0/10	24	1.54		SN	0/10	24	1.00		SN	0/10	24	1.84
	SD _{24h}	0/10	24	1.06		SD _{24h}	5/10	13.38	2.19		SD _{24h}	0/10	24	1.69		SD _{24h}	2/10	20.73	1.33		SD _{24h}	2/10	22.34	0.93		SD _{24h}	0/10	24	1.84
	SD _{48h}	0/10	48	—		SD _{48h}	10/10	17.95	—		SD _{48h}	0/10	48	—		SD _{48h}	4/10	35.36	—		SD _{48h}	2/10	41.54	—		SD _{48h}	0/10	48	—
	SK _{24h}	0/10	24	1.06		SK _{24h}	3/10	22.56	3.69		SK _{24h}	1/10	22.55	1.59		SK _{24h}	0/10	24	1.54		SK _{24h}	0/10	24	1.00		SK _{24h}	1/10	23.75	1.82
SK _{48h}	0/10	48	—	SK _{48h}	3/10	39.36	—	SK _{48h}	1/10	44.15	—	SK _{48h}	0/10	48	—	SK _{48h}	0/10	48	—	SK _{48h}	1/10	45.35	—	SK _{48h}	1/10	45.35	—		
12	SP	8/10	13.92	—	25	SP	8/10	6.12	—	36	SP	10/10	4.18	—	48	SP	9/10	17.81	—	60	SP	9/10	15.04	—					
	SN	0/10	24	1.72		SN	5/10	19.84	3.25		SN	0/10	24	5.74		SN	0/10	24	1.35		SN	1/10	22.73	1.51					
	SD _{24h}	0/10	24	1.72		SD _{24h}	4/10	15.69	2.57		SD _{24h}	8/10	9.29	2.22		SD _{24h}	4/10	17.67	0.99		SD _{24h}	5/10	16.47	1.09					
	SD _{48h}	0/10	48	—		SD _{48h}	5/10	27.90	—		SD _{48h}	10/10	10.72	—		SD _{48h}	8/10	27.49	—		SD _{48h}	10/10	21.08	—					
	SK _{24h}	0/10	24	1.72		SK _{24h}	2/10	22.82	3.73		SK _{24h}	0/10	24	5.74		SK _{24h}	1/10	22.41	1.26		SK _{24h}	0/10	24	1.60					
SK _{48h}	0/10	48	—	SK _{48h}	4/10	38.86	—	SK _{48h}	0/10	48	—	SK _{48h}	3/10	42.54	—	SK _{48h}	0/10	48	—										

TABLE VI
ABLATION RESULTS FOR SYZPILOT VARIANTS.

Metrics Group	Model Performance					Reachability Performance		Reproduction Performance		
Fuzzer Variant	μ TTIS	μ TTFS	Accuracy	Macro-F ₁	Weighted-F ₁	$\text{avg}(\overline{N}_h)$	$\text{avg}(\mu\text{TTH})$	$\text{avg}(\mu\text{TTE})$	Targets (P_t^c)	Runs (P_r^c)
SYZPILOT	0.19h	1.83h	91.94%	0.79	0.87	4,073	0.65h	7.56h	5/5 (100.00%)	45/50 (90.00%)
SYZPILOT _{RandomWaypoint}	0.31h	6.10h	79.33%	0.59	0.63	293	2.62h	21.78h	2/5 (40.00%)	9/50 (18.00%)
SYZPILOT _{NoPred}	-	-	-	-	-	248	5.65h	22.25h	2/5 (40.00%)	8/50 (16.00%)
SYZPILOT _{NA-ALL}	1.22h	7.83h	78.46%	0.62	0.70	106	4.20h	19.58h	3/5 (60.00%)	13/50 (26.00%)
SYZPILOT _{NA-IA}	0.17h	2.93h	87.18%	0.70	0.81	813	1.22h	15.12h	3/5 (60.00%)	22/50 (44.00%)
SYZPILOT _{NA-SA}	0.19h	2.76h	87.15%	0.72	0.81	1,201	0.92h	14.41h	4/5 (80.00%)	26/50 (52.00%)
SYZPILOT _{NA-CS}	1.35h	4.04h	88.23%	0.75	0.82	2,672	2.47h	10.89h	5/5 (100.00%)	33/50 (66.00%)

SYZPILOT_{NoPred} has no model-performance metrics because it does not train or serve a reachability predictor.

comes: SYZPILOT succeeds but is slower than a baseline, at least one baseline succeeds while SYZPILOT does not, and no evaluated fuzzer succeeds. The first outcome can arise when static syscall-entry guidance is already effective for a shallow or weakly state-dependent target, leaving less room for online guidance to reduce exposure time. Bugs #26 and #51 illustrate the second outcome: SYZPILOT reaches both targets in all ten runs but does not expose either UAF because their triggers require cross-task object lifecycles that the current guidance does not explicitly model. The third outcome may involve complex payload, pseudo-syscall, or subsystem-state constraints that all evaluated fuzzers struggle to satisfy reliably. Appendix A-B provides detailed case analysis.

Beyond reproducing historical bugs, separate directed target-only campaigns using externally selected sites also found five previously unknown bugs, including one assigned a CVE. Table VII summarizes these findings.

TABLE VII

UNKNOWN BUGS DISCOVERED BY SYZPILOT DURING THE EXPERIMENTS.

Kernel	Bug Type	Bug Function	Status
6.12	UAF	ocfs2_check_dir_entry	Confirmed
6.12	GPF	btrfs_lookup_csum	Fixed ^C
6.14	kernel BUG	__vb2_perform_fileio	Reported
6.14	NPD	squashfs_cache_delete	Fixed
6.14	NPD	const_folio_flags	Reported

^C: Assigned CVE.

D. Ablation Study

As introduced in §III, SYZPILOT has three main components: waypoint extraction, reachability prediction, and attribution-guided scheduling. We evaluate their contributions by disabling or replacing one component at a time and measuring model performance, target reachability, and bug reproduction. Because attribution guidance has two sources, model interpretability and statistical sequence mining, we separate them in the ablation. This gives six variants: random waypoint selection (SYZPILOT_{RandomWaypoint}), no reachability prediction (SYZPILOT_{NoPred}), no attribution guidance (SYZPILOT_{NA-ALL}), no interpretability-based attribution (SYZPILOT_{NA-IA}), no statistics-based attribution (SYZPILOT_{NA-SA}), and no cold-start syscalls (SYZPILOT_{NA-CS}). We run all variants on five bugs (#17,

#21, #25, #36, and #61) that the full SYZPILOT can reproduce, which makes end-to-end degradation observable. All variants use the same fuzzing configuration and ten 24-hour runs.

Performance of Reachability Prediction Model. We use average time-to-initial-stage (μ TTIS) and average time-to-final-stage (μ TTFS) to measure the collection of initial positive samples and the transition to the final waypoint-level predictor. Table VI also reports accuracy (equivalent to micro-F₁), macro-F₁, and weighted-F₁ to distinguish aggregate from per-class performance under the long-tailed waypoint-label distribution [58]. The full SYZPILOT reaches the final stage fastest at 1.83h and obtains the highest model scores: 91.94% accuracy, 0.79 macro-F₁, and 0.87 weighted-F₁. Random waypoints increase μ TTFS to 6.10h and reduce macro-F₁ to 0.59. Removing cold-start syscalls has the clearest early-stage effect, increasing μ TTIS from 0.19h to 1.35h and μ TTFS to 4.04h. Removing all attribution gives the longest μ TTFS at 7.83h, while removing either attribution source alone produces smaller changes. These results show that the components affect different stages of online training, although the aggregate ablation does not isolate the cause of each model-quality change.

Performance of Bug Reproduction. Table VI also reports end-to-end results on the same five bugs. The three average columns are arithmetic means across targets. Target- and run-level exposure rates use 5 and 50 as their respective denominators. The full SYZPILOT records 4,073 average target-hit executions, 0.65h $\text{avg}(\mu\text{TTH})$, 7.56h $\text{avg}(\mu\text{TTE})$, and 45/50 successful exposure runs. Random waypoints and removing the predictor give the lowest exposure-run counts, 9/50 and 8/50, and increase $\text{avg}(\mu\text{TTE})$ to 21.78h and 22.25h. Removing all attribution reduces $\text{avg}(\overline{N}_h)$ from 4,073 to 106, increases $\text{avg}(\mu\text{TTH})$ from 0.65h to 4.20h, and lowers target- and run-level exposure from 5/5 and 45/50 to 3/5 and 13/50, respectively. Removing either attribution source alone produces a smaller degradation than removing both. These results support the system-level utility of attribution guidance without implying that individual attributed features are causally necessary for exposure. Without cold-start syscalls, SYZPILOT still reproduces all five bugs at least once, but exposure runs decrease to 33/50 and $\text{avg}(\mu\text{TTE})$ increases to 10.89h. Overall, these modules contribute to bug reproduction at different stages rather than through a single shared effect.

E. Performance Overhead

Efficiency is a practical concern for SYZPILOT because it uses both target-specific preprocessing and a GPU-side brain. Table VIII separates these costs by when they are paid and whether they block the fuzzer loop. The offline representation cost is paid once. We collect 2.4 million syz-programs and coverage from three Linux LTS kernels for training. Training the SyzTokenizer with 16 CPU workers takes about 10 wall-clock minutes, and pretraining SyzEncoder for three epochs takes about 16 wall-clock hours on 2×NVIDIA A800 GPUs. This cost is shared by all targets and is not repeated in per-target fuzzing campaigns.

The per-target preprocessing cost is lightweight. SYZPILOT extracts report-derived waypoints through symbol-table lookup and targeted partial disassembly, which takes about one minute per target and does not require the compile-analyze-recompile workflow used by SyzDirect. The Path Analyzer for cold-start syscall inference takes less than one second. We therefore treat these two steps as setup costs before fuzzing starts, rather than as recurring costs inside the fuzzing loop.

The online learning and attribution costs are paid on SYZPILOT-Brain. A single target task uses one A800 GPU for predictor training, with about 20–34GB memory consumption. Exporting one predictor version takes 15–25 wall-clock minutes, and attribution analysis takes less than five minutes. The two A800 GPUs in our deployment are used for load balancing and concurrent tasks, not because one task requires two GPUs. These operations run asynchronously. They do not stop the fuzzer, which keeps using the current model until a new version is deployed.

The direct fuzzer-side overhead comes from data transfer and inference. The average gRPC round-trip time for a batch is about 30ms, and the average inference latency is about 50ms. With the asynchronous input queue, the accumulated blocking cost is 0.15 wall-clock hours (0.63% of a 24-hour campaign), or 0.3 CPU hours with two allocated vCPUs.

TABLE VIII
COST BREAKDOWN OF SYZPILOT.

Component	Scope	Cost	Blocks
Tokenizer	One-time	10 min, 16 CPU workers	No
SyzEncoder	One-time	16 h, 2×A800	No
Waypoints	Per target	~1 min, CPU	Before
Cold-start syscalls	Per target	<1 s, CPU	Before
Predictor training	Per version	15–25 min, 1×A800	No
Attribution	Per version	<5 min, 1×A800	No
RPC and inference	Per batch	30 ms RTT, 50 ms infer.	Yes

VI. DISCUSSION

Why not use LLMs? SYZPILOT uses a compact *SyzEncoder* with a small classification head instead of an LLM for runtime prediction. The reason is practical. Directed kernel fuzzing requires frequent inference and online updates during a 24-hour campaign, while LLM fine-tuning or high-latency inference would interfere with fuzzing throughput [57]. We therefore reserve large models for non-runtime or asynchronous tasks,

such as refining waypoint candidates or analyzing attribution results.

Use Cases and Waypoint Inputs. SYZPILOT supports report-driven reproduction and target-only directed testing. Report-driven mode uses the path context in a Syzkaller-style report to recreate a known failure or test a patch when no stable reproducer is available. Because such reports are normally available for historical syzbot bugs, report-driven mode remains the default. Target-only mode is a fallback for patch testing or directed exploration when only an externally selected function or source location is available. It constructs graph-derived waypoints from this target and a matching kernel call graph.

We evaluate both modes on the five targets used in the ablation study. Target-only mode reaches every target at least once, succeeding in 43/50 reaching runs, but exposes only four bugs in 22/50 runs. Report-derived mode reaches the targets in all 50 runs and exposes all five bugs in 45/50 runs. For Bug #61, target-only mode reaches the target in all ten runs but never exposes the bug, illustrating that graph-derived waypoints provide weaker guidance about the bug-triggering context. Appendix A-D reports the waypoint construction and per-target results.

Synergy Effects. The centralized brain allows limited reuse across fuzzing campaigns. For the same target, a new campaign can warm-start from a previously trained predictor and dataset. In our warm-start experiment, this reduces target-reaching time by 3.2× on average. Across different targets, reuse is more conservative. The SyzEncoder can be shared directly, while target-specific predictors or waypoint labels should only be reused when the tasks share compatible waypoint semantics. SYZPILOT can also consume external startup knowledge, such as syscall entries or templates inferred by SyzDirect or syscall dependencies inferred by G-Fuzz [14].

Limitations. ① *Report and path dependence.* Report-derived waypoints make SYZPILOT effective for bug reproduction and patch testing with known reports, but they bias exploration toward the observed crash path. This is useful when the goal is to recreate the reported failure, but it may miss alternative contexts that reach the same target through different subsystem routes. Target-only mode removes the report dependency but requires an externally selected site and a matching call graph. Missing or spurious call-graph edges can change the selected chain, which does not encode the complete bug-triggering state. Reaching the selected site therefore does not imply bug exposure. ② *Argument-level guidance.* SYZPILOT computes attribution scores over input tokens, but its IG-based guidance acts primarily on syscalls and execution properties. Statistical templates may retain recurring syscall orderings, but they do not provide dedicated guidance for field-level payload construction, pseudo-syscall wrapping or dependencies across complex program states. The case studies in Appendix A-B illustrate how these constraints can separate target reaching from bug exposure. ③ *Online data bias.* The reachability predictor is trained from data collected during the current fuzzing campaign. If early seeds overrepresent one region,

the predictor may learn a narrow view of reachability. The scheduler keeps probabilistic exploration to reduce this risk, but adaptive waypoint refinement and confidence calibration remain open problems. ④ *Resource assumptions.* SYZPILOT assumes access to a GPU-side brain for online training and attribution. As shown in §V-E, these costs do not block the fuzzer loop, and we report compensated 48-hour SyzDirect and Syzkaller runs to account for hardware asymmetry. A CPU-only deployment would likely need less frequent model updates.

Future Work. Future work will focus on making the guidance less path-dependent and more object-aware. First, graph-derived waypoints can be combined with runtime feedback so that SYZPILOT starts from subsystem-level guidance and switches to report-derived guidance once a crash is observed. Second, guidance can be extended from syscall level to argument and object levels, allowing the scheduler to prioritize field-level mutations (such as in netlink messages and ioctl payloads) while preserving dependencies across related syscalls and pseudo-syscalls. Third, state, def-use, and data coverage feedbacks [32], [33] could distinguish executions within the same waypoint and refine online labels or corpus retention. Fourth, cross-task reuse can be made safer by sharing only encoder weights or data associated with compatible waypoints. LLM agents may help in these asynchronous tasks, but they are not a practical replacement for the runtime predictor.

VII. CONCLUSION

In this paper, we presented SYZPILOT, a target-adaptive framework for directed kernel fuzzing. SYZPILOT combines waypoint-guided online reachability prediction with attribution-guided scheduling, allowing the fuzzer to learn target-specific progress and then use attributions from the predictor and recurring syscall patterns to guide mutation and promote bug exposure. By distinguishing target reachability from bug reproduction, SYZPILOT addresses a practical gap in directed kernel fuzzing. Our evaluation on a 70-bug benchmark shows that this design improves both target reaching and bug reproduction compared with state-of-the-art methods.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive feedback. This work is partially supported by NSFC (No. U24A20236, 62302497), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118).

ETHICAL CONSIDERATIONS

SYZPILOT is designed and evaluated as a defensive security research system for kernel bug reproduction, patch testing, and directed testing. The work does not involve human subjects, and we did not collect personally identifiable information. All fuzzing campaigns were conducted in isolated QEMU virtual machines under our control, without interacting with production systems, public networks, or third-party services.

We strictly adhered to responsible disclosure policies for 0-day vulnerabilities discovered by our framework. The main expected benefit of this work is to help kernel maintainers and security researchers validate reported bugs, reproduce failures, and test patches in a controlled setting.

As with other fuzzing tools, open release may lower the cost of reproducing vulnerable behavior. We mitigate this risk by releasing SYZPILOT for legitimate security research, documenting its intended defensive use, and sharing artifacts that support reproducibility without targeting deployed systems.

REFERENCES

- [1] D. Zhan, Z. Yu, X. Yu, H. Zhang, and L. Ye, “Shrinking the kernel attack surface through static and dynamic syscall limitation,” *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1431–1443, 2022.
- [2] Y. Sun, Y. Kang, C. Wu, K. Lu, J. Wang, X. Li, Y. Hu, J. Ren, Y. Lai, M. Xie, and Z. Wang, “Syzparam: Incorporating runtime parameters into kernel driver fuzzing,” in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’25, 2025, p. 1484–1498.
- [3] M. Nadim, W. Lee, and D. Akopian, “Kernel-level rootkit detection, prevention and behavior profiling: A taxonomy and survey,” *arXiv preprint arXiv:2304.00473*, 2023.
- [4] B. Ruan, J. Liu, C. Zhang, and Z. Liang, “Kernjc: Automated vulnerable environment generation for linux kernel vulnerabilities,” in *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, 2024, pp. 384–402.
- [5] H. Li, H. Zhang, K. Pei, and Z. Qian, “Towards more accurate static analysis for taint-style bug detection in linux kernel,” in *40th IEEE/ACM International Conference on Automated Software Engineering, ASE*, 2025.
- [6] R. Helmke and J. vom Dorp, “Extended abstract: Towards reliable and scalable linux kernel cve attribution in automated static firmware analyses,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer Nature Switzerland, 2023, pp. 201–210.
- [7] Google, “Syzkaller,” <https://github.com/google/syzkaller>.
- [8] Google, “Syzbot,” <https://syzkaller.appspot.com/upstream>.
- [9] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, “Semfuzz: Semantics-based automatic generation of proof-of-concept exploits,” in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2139–2154.
- [10] X. Li, Y. Kang, C. Wu, D. Liu, J. Wang, Y. Sun, Z. Wu, Y. Wang, R. Ma, and Q. Wei, “Yesterday once more: facilitating linux kernel bug reproduction via reverse fuzzing,” *IEEE Transactions on Information Forensics and Security*, vol. 20, pp. 5224–5239, 2025.
- [11] X. Du, G. Zheng, K. Wang, Y. Zou, Y. Wang, W. Deng, J. Feng, M. Liu, B. Chen, X. Peng *et al.*, “Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag,” *ACM Transactions on Software Engineering and Methodology*, vol. 35, no. 10, pp. 1–25, 2026.
- [12] M. Lin and H. Chen, “Specauditor: Generating audit specifications for llm-driven bug detection,” in *2026 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2026, pp. 3396–3413.
- [13] E. Geretto, A. Jemmett, C. Giuffrida, and H. Bos, “Libafgo: Evaluating and advancing directed greybox fuzzing,” in *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2025, pp. 355–373.
- [14] Y. Li, Y. Chen, S. Ji, X. Zhang, G. Yan, A. X. Liu, C. Wu, Z. Pan, and P. Lin, “G-fuzz: A directed fuzzing framework for gvisor,” *IEEE Transactions on Dependable and Secure Computing*, 2023.
- [15] X. Tan, Y. Zhang, J. Lu, X. Xiong, Z. Liu, and M. Yang, “Syzdirect: Directed greybox fuzzing for linux kernel,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1630–1644.
- [16] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, “Beacon: Directed grey-box fuzzing with provable path pruning,” in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 36–50.
- [17] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, “Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning,” in *29th USENIX security symposium (USENIX Security 20)*, 2020, pp. 2255–2269.

- [18] S. Gong, D. Peng, D. Altinbükten, P. Fonseca, and P. Maniatis, "Snowcat: Efficient kernel concurrency testing using a learned coverage predictor," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 35–51.
- [19] J. Xu, X. Zhang, S. Ji, Y. Tian, B. Zhao, Q. Wang, P. Cheng, and J. Chen, "Mock: Optimizing kernel fuzzing mutation with context-aware dependency," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.
- [20] S. Gong, W. Rui, D. Altinbükten, P. Fonseca, and P. Maniatis, "Snowplow: Effective kernel fuzzing with a learned white-box test mutator," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 1124–1138.
- [21] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2329–2344.
- [22] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2095–2108.
- [23] Z. Du, Y. Li, Y. Liu, and B. Mao, "Windranger: A directed greybox fuzzer driven by deviation basic blocks," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2440–2451.
- [24] T. E. Kim, J. Choi, K. Heo, and S. K. Cha, "Daft: Directed grey-box fuzzing guided by data dependency," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4931–4948.
- [25] P. Lin, P. Wang, X. Zhou, W. Xie, G. Zhang, and K. Lu, "Deepgo: Predictive directed greybox fuzzing," in *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*, 2024.
- [26] Y. Chen, C. Zhang, L. Wang, W. Zhu, C. Luo, N. Gui, Z. Ma, X. Zhang, and B. Su, "Idfuzz: Intelligent directed grey-box fuzzing," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 6219–6238.
- [27] P. Srivastava, S. Nagy, M. Hicks, A. Bianchi, and M. Payer, "One fuzz doesn't fit all: Optimizing directed fuzzing via target-tailored program state restriction," in *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC)*, 2022, pp. 388–399.
- [28] Y. Xiang, X. Zhang, P. Liu, S. Ji, H. Liang, J. Xu, and W. Wang, "Critical code guided directed greybox fuzzing for commits," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 2459–2474.
- [29] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "StateFuzz: System call-based state-aware linux driver fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3273–3289.
- [30] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data flow sensitive fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2577–2594.
- [31] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "FuzzFactory: Domain-specific fuzzing with waypoints," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.
- [32] M. Wang, J. Liang, C. Zhou, Z. Wu, J. Fu, Z. Su, Q. Liao, B. Gu, B. Wu, and Y. Jiang, "Data coverage for guided fuzzing," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 2511–2526.
- [33] L. Weiß, T. Holl, and K. Borgolte, "Storfuzz: Using data diversity to overcome fuzzing plateaus," in *Proc. Int. Conf. Software Engineering (ICSE)*, M. Mezini and T. Zimmermann, Eds. Association for Computing Machinery (ACM)/Institute of Electrical and Electronics Engineers (IEEE), 2026.
- [34] C. Liao, R. Wang, Y. Li, J. Chen, Y. Li, and Z. Pan, "Syzlego: Enhancing kernel directed greybox fuzzing via dependency inference and scheduling," in *International Conference on Information Security*. Springer, 2024, pp. 171–189.
- [35] Y. Shen, S. Chen, J. Liu, Y. Xu, Q. Zhang, R. Wang, H. Shi, and Y. Jiang, "Brief industry paper: Directed kernel fuzz testing on real-time linux," in *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2023, pp. 495–499.
- [36] H. Shi, S. Chen, R. Wang, Y. Chen, W. Zhang, Q. Zhang, Y. Shen, X. Shi, C. Hu, and Y. Jiang, "Industry practice of directed kernel fuzzing for open-source linux distribution," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2159–2169.
- [37] Y. Shen, J. Liu, Y. Chen, Q. Zhang, R. Wang, H. Shi, and Y. Jiang, "Directed real-time linux fuzzing with configuration awareness," *ACM Transactions on Embedded Computing Systems*, 2025.
- [38] M. Fleischer, D. Das, P. Bose, W. Bai, K. Lu, M. Payer, C. Kruegel, and G. Vigna, "Actor: Action-guided kernel fuzzing," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5003–5020.
- [39] B. Johannesmeyer, R. Iseman, C. Giuffrida, and H. Bos, "Dynamic detection of vulnerable dma race conditions," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 4499–4513.
- [40] Z. Lin, Y. Chen, Y. Wu, D. Mu, C. Yu, X. Xing, and K. Li, "Grebe: Unveiling exploitation potential for linux kernel bugs," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2078–2095.
- [41] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, "Syzscope: Revealing high-risk security impacts of fuzzer-exposed bugs in linux kernel," in *31st USENIX security symposium (USENIX Security 22)*, 2022, pp. 3201–3217.
- [42] M. Yuan, B. Zhao, P. Li, J. Liang, X. Han, X. Luo, and C. Zhang, "Ddrace: Finding concurrency uaf vulnerabilities in linux drivers with directed fuzzing," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2849–2866.
- [43] S. Bai, Z. Zhang, and H. Hu, "Countdown: Refcount-guided fuzzing for exposing temporal memory errors in linux kernel," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1315–1329.
- [44] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. Abu-Ghazaleh, "Syzvegas: Beating kernel fuzzing odds with reinforcement learning," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2741–2758.
- [45] C. Yang, Z. Zhao, and L. Zhang, "Kernelgpt: Enhanced kernel fuzzing via large language models," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 560–573.
- [46] Z. Zhang, L. Li, R. Liang, and K. Chen, "Unlocking low frequency syscalls in kernel fuzzing with dependency-based rag," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 848–870, 2025.
- [47] X. Li, Z. Yuan, Z. Zhang, Y. Sun, and L. Zhang, "Towards large language model guided kernel direct fuzzing," in *International Conference on Fundamental Approaches to Software Engineering*. Springer Nature Switzerland Cham, 2025, pp. 33–42.
- [48] G. Li, M. Sridharan, and Z. Qian, "Redefining indirect call analysis with kallgraph," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 2957–2975.
- [49] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *International conference on machine learning*. PMLR, 2017, pp. 3319–3328.
- [50] C. Li, Q. Yang, J. Wang, and M. Li, "Efficient mining of gap-constrained subsequences and its various applications," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 6, no. 1, pp. 1–39, 2012.
- [51] B. Iglewicz and D. C. Hoaglin, *Volume 16: how to detect and handle outliers*. Quality Press, 1993.
- [52] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcode: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.
- [53] D. Zhang, W. U. Ahmad, M. Tan, H. Ding, R. Nallapati, D. Roth, X. Ma, and B. Xiang, "Code representation learning at scale," in *The Twelfth International Conference on Learning Representations*, 2024.
- [54] BigCode, "Starencoder model card," <https://huggingface.co/bigcode/starencoder>.
- [55] "TorchServe," <https://github.com/pytorch/serve>.
- [56] "SyzDirect Dataset," <https://github.com/seclab-fudan/SyzDirect/tree/main/dataset>.
- [57] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "Sok: Prudent evaluation practices for fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1974–1993.
- [58] J. Opitz, "A closer look at classification evaluation metrics and a critical reflection of common evaluation practice," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 820–836, 2024.

A. Open Science

To support reproducibility and future research, we open-source our prototype, including the source code for SYZPILOT-Fuzzer, SYZPILOT-Brain, the pre-trained SyzEncoder model, and the benchmark datasets.

Our public repository is available at <https://github.com/QGrain/SyzPilot>, with links to the released model and datasets. The persistent DOI resolved to our latest archived release is also available at <https://doi.org/10.5281/zenodo.22874327>.

B. Bug Reproduction Case Studies

Successful reproductions. We use Bug #1 to compare the report-derived progression with its independently documented trigger. This comparison checks whether the supervision and mutation vocabulary cover the known triggering context rather than assigning a causal role to any attributed token.

Bug #1 is a checksum-offset BUG in linux kernel net subsystem (`validate_xmit_skb`). Its PoC creates and configures a raw packet socket, installs the `cake` qdisc through `sendmsg$nl_route_sched`, and invokes `sendto$inet6` with a 108-byte initialized packet buffer and length `0x3287`. The nine retained waypoints capture the nested transmit path from `packet_snd` through the outer `__dev_queue_xmit`, `sch_direct_xmit`, `geneve_xmit`, and the inner `__dev_queue_xmit`. PoC replay covers five of their PCs. These intermediate labels provide target-specific progress along the nested path and supply waypoint-reaching programs to the attribution and sequence guidance. SyzDirect uses static distance, Snowplow learns mutation positions for general coverage, and Syzkaller uses coverage-guided mutation, but none updates mutation guidance from online target-specific progress in this evaluation. SYZPILOT reaches and reproduces the bug in 10/10 runs with a mean TTE of 10.33 hours, whereas each baseline records 0/10 reproductions.

Underperformance and failures. We organize the analysis by observed outcome: cases where SYZPILOT reproduces the bug but is slower than a baseline, cases reproduced only by another fuzzer, and cases that no evaluated fuzzer reproduces. We report the 24-hour comparisons separately from the compensated 48-hour baseline runs.

Bug #26 reaches `__kernfs_remove` in 10/10 SYZPILOT runs with a mean TTH of 0.02 hours, but no run exposes the UAF. Its repeated six-process PoC constructs a `tmpfs` mount and `fd`-backed `9p` connection before concurrent tasks remove and access the same `kernfs` root. The fixing commit `4abc99652812` confirms that concurrent `kernfs_remove_by_name_ns` calls can free this root during `kernfs_drain`. SYZPILOT repeatedly reaches the removal path, but its syscall, property, and sequence guidance does not represent the inter-task reference lifetime required by the race. Snowplow and Syzkaller expose the bug in only 1/10 and 2/10 24-hour runs, respectively, and the 48-hour Syzkaller group remains at 2/10, indicating that their general exploration also satisfies the lifecycle only rarely.

Bug #51 similarly reaches its target in 10/10 SYZPILOT runs with a mean TTH of 2.69 hours but yields 0/10 reproductions. The PoC asynchronously invokes `unshare(0x6c060000)`, then creates an IPv6 TCP socket and calls `sendto$inet6`. The `unshare` path allocates `struct nfsd_net`, registers `nfsd_client_shrinker`, and frees the namespace object after a later initialization failure without first removing that shrinker. A subsequent `mq_init_ns` path accesses the stale list node through `register_shrinker_prepared`. The fixing commit `bd86c69dae65` adds the missing unregister step. The online guidance identifies target-proximate syscalls but does not encode this failure-and-cleanup ordering. SyzDirect likewise has no explicit lifecycle model, and its increase from 3/10 reproductions at 24 hours to 8/10 at 48 hours indicates that additional exploration time materially affects this case.

In the remaining cases, a baseline can be faster when syscall-entry guidance already suffices for a shallow target. Other failures involve structured payloads, pseudo-syscall helpers, or subsystem setup beyond the final target hit, reflecting limitations at different levels of the current guidance.

C. Analysis of Rule-Level Waypoint Extraction

Aggregate effects. We analyze cross-case changes over all 70 benchmark cases and retain detailed extraction trajectories for 67. Hit Rate and Quality use the 42 cases with compatible PoC KCOV. Table IX presents the five stages described in Section III-B. Call Trace is the unpruned input, and the final Greedy Pruning output includes both budget pruning and its post-pruning outlier cleanup. Address resolution is an implementation step rather than a separate extraction stage.

TABLE IX
EFFECTS OF REPORT-DERIVED WAYPOINT EXTRACTION.

Stage	Changed	Avg. Len.	Hit Rate	Quality
Call Trace	–	20.87	39.79%	44.24
Trace Sanitization	69/70	11.45	41.59%	61.26
Complexity Filtering	57/70	8.21	43.50%	67.02
Trivial Stage Merging	41/70	7.03	44.64%	69.32
Greedy Pruning	31/70	5.85	45.64%	71.30

Quality is an auxiliary score, not an end-to-end reproduction measure.

Changed counts the cases modified by each named stage relative to its immediate input. Complexity Filtering follows the unlisted address-resolution step, while the final row combines budget pruning and outlier cleanup relative to Trivial Stage Merging. Trace Sanitization changes 69/70 cases, followed by Complexity Filtering (57/70), Trivial Stage Merging (41/70), and the final Greedy Pruning output (31/70). The earlier transformations therefore affect more cases than the final stage. Across the retained detailed trajectories, mean length decreases from 20.87 to 5.85, while the micro-aggregated hit rate increases from 39.79% to 45.64%.

Scoring. The auxiliary score is $Q = 0.30S + 0.20H + 0.50L$. The semantic component S combines blinded model-assisted ratings of target fidelity, report-section fidelity, causal coherence, and parsimony with deterministic PC observability.

Method identity and KCOV evidence are hidden from the model-assisted evaluator. For h PoC hits among r nonduplicate operational labels, $H = 50 + 30h/r + 10 \min(h/4, 1) + 10I(\text{target hit})$. For n candidates before output normalization, $L = 100 \min(4/n, 1)$, where 4 is the existing minimum waypoint budget. These weights are used only for this offline descriptive analysis and do not affect extraction, model training, scheduling, or the end-to-end results. PoC KCOV provides lower-bound positive evidence that a waypoint is executable. A miss is inconclusive, and coverage does not establish bug reproduction.

Examples. For Bug #1, the five-stage sequence decreases from 36 to 29, 18, 16, and 9 waypoints, with 5/9 final waypoints covered by the PoC. For Bug #32, it decreases from 29 to 14, 9, 7, and 7, and all seven final waypoints are covered. The unchanged final length in Bug #32 shows that pruning is conditional. The listing reports the final entry-to-target order and resolved 32-bit PCs used by the fuzzer.

Bug #1: entry → target 0x70b2a58 ...sys_send@net/socket.c:2019 0x87ebbb3a packet_snd@net/packet/af_packet.c:3016 0x8714da05 ...dev_queue_xmit@net/core/dev.c:4253 0x8714e856 ...dev_xmit_skb@net/core/dev.c:3944 0x730eb33 sch_direct_xmit@net/sched/sch_generic.c:342 0x84c8951c geneve_xmit@drivers/net/geneve.c:1082 0x87925004 ip_finish_output2@net/ipv4/ip_output.c:230 0x8714d14e ...dev_queue_xmit@net/core/dev.c:4276 0x8714b0d2 validate_xmit_skb@net/core/dev.c:3763	Bug #32: entry → target 0x81ca1303 path_mount@fs/namespace.c:3211 0x81ca14d8 do_new_mount@fs/namespace.c:2881 0x81c14b36 vfs_get_tree@fs/super.c:1496 0x82e4683f v9fs_mount@fs/9p/vfs_super.c:126 0x83d8fb1d kobject_put@lib/kobject.c:753 0x83d8fc62 kobject_cleanup@lib/kobject.c:705 0x8882b23c p9_client_create@net/9p/client.c:1063
--	---

D. Evaluation of Target-Only Mode

Setup. We evaluate target-only mode on the same five targets used in the ablation study: Bugs #17, #21, #25, #36, and #61. These targets form a controlled subset reproduced by the default system rather than a representative sample of all 70 cases. Both modes use the same final target PC, kernel image, resources, initial corpus policy, and 24-hour timeout. Report-derived mode uses the complete default configuration, including report-based waypoint and cold-start guidance. Target-only mode receives only the externally selected target location and matching-build call graph. The comparison therefore evaluates the two complete input modes rather than isolating the waypoint source.

TABLE X
REPORT-DERIVED AND GRAPH-DERIVED WAYPOINTS ON FIVE TARGETS.

ID	Report-derived			Graph-derived		
	Length	Time (s)	Quality	Length	Time (s)	Quality
17	4	36.24	89.10	5	49.84	70.33
21	8	55.12	66.10	2	42.37	53.67
25	7	42.07	63.10	3	27.71	56.67
36	4	41.55	77.49	5	69.87	62.90
61	5	46.32	80.20	5	50.34	63.10

Waypoint construction. We assess waypoint quality using the same offline score defined above and report the retained per-target construction times. For the graph-derived chains, reports and PoC KCOV are withheld during construction and used only by the offline evaluator. Graph-derived time includes loading an existing call graph, entry discovery,

path sampling, filtering, and location resolution, but excludes LLVM IR generation, whole-kernel call-graph construction, and KCOV instrumentation-cache generation. As shown in Table X, graph-derived chains are shorter on average (4.00 versus 5.60 waypoints) and take comparable time to construct (48.03 versus 44.26 seconds per target). Report-derived chains score higher on all five targets under the report-informed auxiliary metric, averaging 75.20 versus 61.33. This score characterizes waypoint construction rather than end-to-end fuzzing effectiveness.

TABLE XI
REPORT-DERIVED AND TARGET-ONLY PERFORMANCE ON FIVE TARGETS.

ID	Mode	Runs	μTTH (h)	$\overline{N_h}$	Runs	μTTE (h)
17	SP_R	10/10	0.78	628	7/10	9.93
	SP_T	10/10	1.65	714	3/10	20.31
21	SP_R	10/10	0.60	290	10/10	0.97
	SP_T	8/10	5.93	189	8/10	7.47
25	SP_R	10/10	0.60	7,981	8/10	6.12
	SP_T	8/10	5.44	2,191	5/10	16.97
36	SP_R	10/10	1.13	5,813	10/10	4.18
	SP_T	7/10	8.73	1,695	6/10	11.78
61	SP_R	10/10	0.12	5,653	10/10	16.59
	SP_T	10/10	0.24	2,735	0/10	24

SP_R and SP_T denote report-derived and target-only modes.

End-to-end performance. We compare target reachability and bug reproduction over ten independent 24-hour runs per target. The report-derived values in Table XI are copied from the main ten-run evaluation. Target-only mode reaches every target at least once and succeeds in 43/50 reaching runs, with $\text{avg}(\mu\text{TTH})$ of 4.40 hours and $\text{avg}(\overline{N_h})$ of 1,505. Report-derived mode succeeds in all 50 runs, with $\text{avg}(\mu\text{TTH})$ of 0.65 hours and $\text{avg}(\overline{N_h})$ of 4,073. For bug exposure, target-only mode succeeds on four targets in 22/50 runs, compared with five targets in 45/50 runs for report-derived mode. Their $\text{avg}(\mu\text{TTE})$ values are 16.11 and 7.56 hours, respectively. In particular, target-only mode reaches Bug #61 in all ten runs but does not expose it, illustrating that graph-derived progress signals do not encode the complete bug-triggering context.

APPENDIX B

ARTIFACT APPENDIX

In the paper, we present SYZPILOT as a target-adaptive framework that steers directed fuzzing via reachability prediction and attribution-guided scheduling. This appendix provides a self-contained roadmap for installing and functionally evaluating the Preparation, SYZPILOT-Fuzzer, and SYZPILOT-Brain phases.

A. Description & Requirements

The artifact contains the Waypoint Extractor and the other Preparation modules, SYZPILOT-Brain, and a patch that extends Syzkaller commit 6e83b42 into SYZPILOT-Fuzzer. It also contains benchmark metadata, a three-case mini-benchmark, evaluation scripts, and a unified Docker recipe. We release the pretrained SyzEncoder and its dataset separately on Hugging Face. The source archive excludes compiled target kernels, target reproducers, and VM images. The unified Docker image contains a disposable QEMU guest template but no precompiled target kernel.

1) *How to access:* The v0.2.0 artifact is archived at <https://doi.org/10.5281/zenodo.23539966> and corresponds to GitHub revision 7d397f1ae3fc. The concept DOI <https://doi.org/10.5281/zenodo.22874327> resolves to the latest archived version. We maintain the source at <https://github.com/QGrain/SyzPilot> and publish the unified image as `qgrain/syzpilot:ndss27-ae`. Its Linux/amd64 image manifest digest has prefix `sha256:72cab9f5551a`, abbreviated to 12 hexadecimal digits for readability. The evaluator-facing guide is <https://github.com/QGrain/SyzPilot/blob/main/artifact/README.md>. The image’s `image-info` command reports the included source revision.

2) *Hardware dependencies:*

- **CPU and storage.** Kernel preparation and E1 need an x86-64 Linux host with at least 8 CPU cores and about 30 GB of free workspace beyond the container image.
- **Virtualization.** E2-A additionally needs KVM for one QEMU VM with two vCPUs and 4 GB of memory.
- **GPUs.** E2-B adds SYZPILOT-Brain. Our evaluator profile uses separate NVIDIA GPUs for training and model serving, with approximately 24 GB and 12 GB of free GPU memory, respectively. The Brain and Fuzzer roles may run on separate private-network hosts.

E2-B requires two-way connectivity between these roles for data transfer, inference, and model-ready callbacks.

3) *Software dependencies:*

- **Host tools.** The evaluation uses Docker and QEMU/KVM. E2-B also needs the NVIDIA container runtime.
- **Bundled environment.** The unified image provides the required Go, Python, Java, PyTorch, TorchServe, Transformers, SyzEncoder, tokenizer, Syzlang, and GCC 11.4 environments.

- **Source installation.** For a source installation, `requirements/requirements-brain.txt` includes the `tree-sitter` packages required by `get_targets.py`.

Evaluators must run vulnerable kernels only inside isolated VMs. The target PoC or reproducer must not be used as a fuzzing input or as guidance.

4) *Benchmarks:* The repository provides metadata for the paper’s 70-bug benchmark. The source-only mini-benchmark contains cases 21, 25, and 36, including their kernel commits, configurations, reports, titles, and integrity hashes. The functional workflow uses case 36. Evaluators compile its matching kernel and resolve target PCs from that build’s `vmlinux`.

B. Artifact Installation & Configuration

We use the same image for the Brain and Fuzzer roles. The image starts no service by default. After pulling `qgrain/syzpilot:ndss27-ae` and creating the container roles, run the following commands from `/root/SyzPilot` inside the Fuzzer container:

```
scripts/artifact_eval.sh image-info
scripts/artifact_eval.sh prepare-fuzzer
scripts/artifact_eval.sh build-case36
```

These commands inspect the bundled revision, patch and build the Fuzzer, and compile the matching case-36 kernel with GCC 11.4, respectively.

The first image pull normally takes 10–30 minutes. Preparing SYZPILOT-Fuzzer takes 5–20 minutes. The reusable kernel build takes about 1–3 hours. The complete commands, mounts, network ports, expected output, and cleanup procedure are provided in `artifact/README.md`. The kernel build can be reused across E1 and both E2 profiles while its source, configuration, compiler, and kernel binaries remain unchanged.

C. Experiment Workflow

E1 evaluates report-derived Preparation on the matching case-36 kernel. E2-A then evaluates the Fuzzer-side execution, KCOV, and target-observation path without starting the Brain. E2-B runs the online feedback loop from labeled program collection through training, model deployment, inference, and guidance. E1 and E2-A can be completed without an NVIDIA GPU. E2-B requires the two GPU roles described above.

D. Major Claims

- (C1): Given a public bug report and the matching kernel build, the Preparation phase extracts report-derived waypoints and resolves nonempty, build-specific target PCs. E1 validates this implementation of the Preparation design described in the paper.
- (C2): SYZPILOT-Fuzzer boots the target VM, executes syz-programs, collects KCOV feedback, and records target-PC observations. E2-A validates this Fuzzer-side execution and observation workflow.
- (C3): In the online workflow, SYZPILOT-Fuzzer registers with SYZPILOT-Brain and streams labeled programs. The Curriculum Trainer trains a target-specific reachability

predictor, the Model Server deploys a versioned model, and the Fuzzer receives predictions and target-specific guidance. E2-B validates this feedback loop.

These are functional claims about the artifact. They do not reproduce the paper’s comparative target-reachability or bug-reproduction results.

E. Evaluation

1) *Experiment (E1): Report-derived Preparation [5 human-minutes + 5–15 compute-minutes after the kernel build].*

[How to] Follow the E1 section of `artifact/README.md`.

[Preparation] Complete the image, Fuzzer, and case-36 kernel setup. The exact `vmlinux` used by the VM must be present.

[Execution] Run `scripts/artifact_eval.sh extract-case36` inside the Fuzzer container.

[Results] A successful run prints an ordered waypoint chain and a nonempty JSON list containing three target PCs. Repeating the experiment with the same source, configuration, compiler, and kernel build must produce the same list. This experiment validates C1.

2) *Experiment (E2-A): Fuzzer-side functional test [5 human-minutes + 10 compute-minutes].*

[How to] Follow E2-A in `artifact/README.md`. This experiment does not start SYZPILOT-Brain and needs no GPU.

[Preparation] Run `prepare-fuzzer-only-config` after E1. The generated manager configuration enables target observation but disables Brain registration and model guidance.

[Execution] Run `start-fuzzer-only`. After the bounded run, run `fuzzer-status`.

[Results] The guest must boot without an immediate fatal exit. The latest bench snapshot must report nonzero total executions, coverage, and executor target observations. A final-target hit or crash is not required in this short functional test. This experiment validates C2.

3) *Experiment (E2-B): Online training and inference test [10–20 human-minutes + up to 4 compute-hours].*

[How to] Follow E2-B in `artifact/README.md`. Use mutually reachable private addresses for the Brain and Fuzzer roles.

[Preparation] Run `prepare-config`, run `verify-brain` in the Brain container, and assign separate visible GPUs to training and model serving.

[Execution] Start SYZPILOT-Brain, then run `start-fuzzer` on the Fuzzer host. Inspect progress with `brain-status`, `fuzzer-status`, and the Controller task endpoint.

[Results] A successful run shows Fuzzer registration, labeled Receiver batches, report-derived guidance, a promoted model version, a model-ready notification, and at least one successful inference request. If these events do not appear within four hours, report E2-B as incomplete. A target-PC hit

or target-bug report is not required. This experiment validates C3.

Evidence and cleanup. Retain the waypoint output, manager bench snapshots, and Fuzzer logs. For E2-B, also preserve the Brain log, Receiver data, model state, and final Controller task view. The documented bind mounts keep these files available after container removal.

F. Customization

Evaluators may select different idle CPU pairs, private host addresses, and run roots through the variables documented in `artifact/README.md`. A repeated invocation with the default run root resumes its saved corpus. For an independent run, set `SYZPILOT_RUN_ROOT` to a fresh directory consistently during waypoint extraction, configuration, and fuzzing. Users can apply SYZPILOT to other targets by providing a public report and matching kernel build, then resolving fresh target PCs from that build. We encourage evaluators and users to consult the latest workflows, configuration options, and usage notes at <https://github.com/QGrain/SyzPilot> before customizing longer campaigns.