KnitFuzz: LLM-guided Kernel Fuzzing via Context-Sensitive Socket System Calls

Siwei Zhang
Syracuse University
Syracuse, NY, USA
szhan197@syr.edu

Endadul Hoque
Syracuse University
Syracuse, NY, USA
enhoque@syr.edu

Abstract

The kernel's syscall interface, particularly for socket-based networking, remains a critical yet complex attack surface. Existing kernel fuzzers, such as Syzkaller, struggle to generate valid test cases for socket syscalls due to their contextual state—maintained across userspace variables and kernel data structures. In this paper, we present KnitFuzz, a novel kernel fuzzing framework that leverages large language models (LLMs) to generate context-aware C programs as seed inputs. Unlike prior fuzzers that depend on Syzkaller's domain-specific language and programs, KnitFuzz focuses on C to avoid domain language-specific constraints and to benefit from the LLMs' prior knowledge of socket-based networking programs. KnitFuzz introduces selective memory instrumentation for effective mutation and incorporates a dynamic, state-aware remote endpoint to simulate realistic network interactions when necessary. We evaluated KnitFuzz on multiple Linux kernels, demonstrating significant improvements over state-of-the-art fuzzers: up to 19.38% higher basic block coverage and 29.84% higher branch coverage. KnitFuzz also uncovers two kernel bugs, including a previously unknown vulnerability. These results highlight KnitFuzz's effectiveness in exercising deep, complex paths in the kernel's network subsystem.

CCS Concepts

• **Security and privacy → Operating systems security**; *Security protocols*; • **Software and its engineering → Software safety**.

Keywords

Kernel Security; Networking; Fuzzing; LLM

ACM Reference Format:
Siwei Zhang and Endadul Hoque. 2026. KnitFuzz: LLM-guided Kernel Fuzzing via Context-Sensitive Socket System Calls. In *Proceedings of the Sixteenth ACM Conference on Data and Application Security and Privacy (CODASPY '26), June 23–25, 2026, Frankfurt am Main, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3800506.3803511

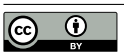
1 Introduction

The kernel is the heart of the operating system (OS), handling hardware management, communications, and I/O—hence, serving as the foundation for system components and userspace applications. Ensuring kernel security is crucial, as vulnerabilities can lead to severe consequences such as privilege escalation (e.g., CVE-2023-0461), unauthorized code execution (e.g., CVE-2017-8824), sandbox escapes (e.g., CVE-2022-31696), and denial-of-service attacks (e.g., CVE-2024-0565)—all of which can result in damages amounting to billions of dollars. Kernel fuzzing has become a powerful automated technique for discovering such vulnerabilities. Notably, Syzkaller [40] has found a lot of bugs over the years. Most kernel fuzzers, including Syzkaller, target the system call (syscall) interface, as syscalls are the primary interaction point between unprivileged user-level programs and the kernel. This interface presents a significant attack surface for exploiting bugs in syscall implementations.

The *socket layer interface* (or *socket API*) is a collection of networking related syscalls that bridges user space with the kernel's networking subsystems. It provides a unified abstraction for accessing diverse protocol families, from traditional TCP/UDP to newer technologies such as RDMA [44] for high-speed remote memory access. Like traditional protocols (e.g., CVE-2020-25705 in TCP), these newer components have also suffered from severe vulnerabilities (e.g., CVE-2020-36385 in RDMA sockets). Given the central role of networking subsystems, thoroughly fuzzing socket-based components in the Linux kernel is essential.

The performance of a fuzzer depends heavily on the quality and diversity of its test cases, which are typically user-space programs invoking target syscalls. Generating such test cases for socket-based syscalls is challenging because they are ***context-sensitive***, with behavior shaped by their implicit state and interdependencies. Without a properly structured sequence of syscalls, consistent parameter values, and correct data flow, test programs are often rejected by the kernel's shallow error-handling logic. This reduces fuzzing efficiency and limits code coverage.

Socket-based syscalls rely strongly on contextual state, maintained across both userspace variables and kernel data structures. As a result, existing kernel fuzzers [18, 19, 36, 40, 70, 85, 86, 90, 98, 103, 105, 106] struggle to generate valid call sequences. This complexity stems from four main factors: **C1-*Abstract Wrappers***. Many POSIX socket syscalls are high-level wrappers whose behavior depends on the context. For example, `listen()` dispatches to `inet_listen()` only if the descriptor from `socket()` is built using `AF_INET` (IPv4). **C2-*Weak Typing***. Socket syscalls often accept loosely typed parameters (e.g., the `void*` in `setsockopt()`), whose interpretation depends on contextual information from other arguments. **C3-*Implicit Dependencies***. Some syscalls, such as `setsockopt()`, silently influence the behavior of subsequent calls like `bind()`, without explicit program-level indicators. **C4-*Remote Endpoint Influence***. Kernel's internal state may depend on responses from remote endpoints. For instance, a TCP client's `send()` must operate on a socket in a *connected* state, established by



CODASPY '26, Frankfurt am Main, Germany

ACM ISBN 979-8-4007-2562-3/2026/06
https://doi.org/10.1145/3800506.3803511

`connect()`. If the remote endpoint is invalid or misbehaves, the `connect()` call fails, preventing subsequent operations like `send()`.

Prior work, Syzkaller [40], generates test cases using manually written syscall specifications. To handle challenges like **C**1 and **C**2, it relies on domain experts to encode submodule-specific syscalls and parameters—a process that is manual, time-consuming, and error-prone. To address this, other fuzzers [18, 19, 70, 85, 98, 103, 105] have explored methods like mutating seed programs or auto-generating syscall specifications. However, their success depends on high-quality seeds and accurate API models. Some [18, 19, 36, 45, 52, 70, 85, 86] infer syscall dependencies via trace or static analysis, but these often yield false positives, limiting their practical utility.

Additionally, kernel fuzzers [18, 19, 36, 52, 70, 85, 86, 90, 98, 103, 105, 106] that rely on Syzkaller's specification language, *Syzlang* [41], struggle with **C**3, as Syzlang cannot express implicit dependencies between syscalls—key to exercising deeper and more complex syscall sequences. While explicit dependencies can be specified and enforced, implicit ones are left to chance, often resulting in poorly explored paths [105] or invalid test cases.

Although the interactions between the host running the target kernel and the remote endpoint can help the fuzzer explore meaningful paths and reach deeper into system call logic (**C**4), most fuzzers, including Syzkaller [16, 18, 19, 36, 40, 52, 70, 85, 86, 90, 98, 103, 105, 106], do not directly use remote endpoints. Instead, they emulate packet delivery from a remote endpoint by embedding domain-specific (hook) functions into test programs. However, these emulations are largely uncontrolled, often producing responses irrelevant to the target protocol, which derails the exploration of deeper paths and reduces efficiency.

In this paper, we present **KnitFuzz** (Kernel Network InTerface FUZZer), a kernel fuzzer designed to test socket-based Linux networking subsystems by addressing challenges **C**1–**C**4. Our key insight is that large language models (LLMs) [8, 42, 68] can be leveraged in a principled way to reduce the human effort traditionally required to guide fuzzers. KnitFuzz operates in two stages: (a) generating context-aware seed programs that invoke socket syscalls, and (b) fuzzing the target kernel using these seeds. Each stage is designed to run independently, allowing seed generation to be precomputed without hindering fuzzing throughput or progress.

While LLMs are powerful, they cannot solve the end-to-end problem of kernel fuzzing alone. As Apple's recent AI Illusion paper [82] highlights, LLMs often give the appearance of reasoning without reliably handling complex, context-sensitive tasks. To this end, KnitFuzz leverages LLMs where feasible, but complements them with new techniques for generating context-aware seed programs, performing context-preserving mutations, and constructing subsystem-specific remote endpoints.

KnitFuzz addresses **C**1–**C**3 by utilizing the nuanced knowledge embedded in LLMs trained on large corpora of source code and syscall usages, including socket-related APIs. It builds an automated pipeline that constructs *zero-shot* prompts with several socket syscalls, queries the LLM for context-aware candidate C programs, evaluates their validity, and refines them as needed. Unlike prior work [105], which uses LLMs to generate Syzkaller programs (*syz-programs*) through in-context learning of domain-specific syntax, KnitFuzz generates C programs directly, avoiding language-specific constraints.

Beyond initial seeds, effective fuzzing requires a steady supply of test programs to improve kernel code coverage. Conventional strategies such as seed mutation or random program generation—employed by existing fuzzers [40, 70, 103, 105]—often degrade efficiency by violating the context-sensitivity of original programs. To address this, KnitFuzz introduces argument-level mutation techniques that preserve both data-flow and control-flow consistency.

Because KnitFuzz generates user-level C programs as seeds, it cannot directly reuse Syzkaller's fuzzing engine, unlike prior efforts [18, 19, 70, 85, 98, 103, 105]. Instead, it builds a custom testbed around a user-space coverage-guided fuzzer (AFL++ [31]), while using it to *indirectly* fuzz the kernel. Each seed program is executed inside a virtual machine running the target kernel, where AFL++ is customized to (i) focus on kernel-level coverage instead of user-level coverage, and (ii) mutate only the instrumented memory locations exposed by generated seeds. To further enhance exploration, KnitFuzz incorporates a dynamic, state-aware remote endpoint that addresses **C**4 by enabling protocol-compliant network interactions.

We implemented a fully functional fuzzer based on KnitFuzz, with components developed in Python 3.12, C, C++ on LLVM 13, and Go. We evaluated KnitFuzz on several socket-related networking subsystems in Linux v6.12.42 (LTS), focusing on protocol domains such as `AF_INET`, `AF_INET6`, `AF_UNIX`, `AF_PACKET`, and `AF_SMC` (Shared Memory Communication over RDMA). Our experiments first empirically tuned key system parameters to optimize seed generation, and then compared KnitFuzz against prior fuzzers. KnitFuzz explored 57785 branches, which is 14.12% higher branch coverage than Syzkaller, 14.59% more than Moonshine [70], and 29.84% more than Healer [86]. Furthermore, KnitFuzz uncovered two bugs, demonstrating its practical effectiveness in exposing real-world kernel vulnerabilities.

In summary, we make the following contributions:

- We present KnitFuzz, an end-to-end kernel fuzzer that leverages LLMs to test Linux networking subsystems.
- KnitFuzz introduces an automated pipeline to generate context sensitive C seed programs for socket-based subsystems, context preserving argument-level mutation techniques, and dynamic remote endpoints that enable access to previously unreachable code paths.
- We implemented and evaluated KnitFuzz on Linux. KnitFuzz improves basic block coverage by 19.13%–19.38% and branch coverage by 14.12%–29.84% over state-of-the-art fuzzers, and it discovers two bugs, all of which remain unpatched. KnitFuzz is available at https://github.com/syne-lab/knitfuzz.

2 Preliminaries

Fuzzing. Fuzzing is a security-focused software testing technique in which a fuzzer automatically generates inputs to explore different execution paths, aiming to uncover abnormal behavior such as crashes. Such behavior typically signals bugs that may be exploitable maliciously [28]. To facilitate fuzzing, programs are often instrumented with sanitizers, which cause crashes upon detecting unsafe behavior [37, 81, 84], and with information-gathering modules that record runtime metrics such as *code coverage* to evaluate fuzzing performance.

Guided by this feedback, fuzzers generate new inputs by mutating valuable inputs in the corpus, which has previously improved

metrics (e.g., branch coverage). A *fuzzing campaign* consists of repeatedly running the target program with different inputs, observing execution, and flagging suspicious behaviors until a terminating condition is reached (e.g., time limit or lack of new bugs). Because typically fuzzers concretely execute binaries, they are largely independent of the target's implementation language and can be easily scaled with additional computing resources.

For example, the popular coverage-guided fuzzer AFL [101] instruments programs at compile time. During execution, the instrumented code tracks transitions between basic blocks, computes unique identifiers for each transition, and reports them to the fuzzer to guide subsequent input generation, enabling systematic exploration of the program's code paths.

Kernel Fuzzing. In contrast, kernel fuzzers [16, 18, 19, 36, 40, 45, 52, 56, 62, 70, 85, 86, 90, 95, 98, 103, 105, 106] focus on testing the kernel via the system call interface, often using test programs that invoke one or more system calls. Building on this idea, coverage-guided kernel fuzzers [57, 75, 79, 92] often adapt AFL [101], a user-space fuzzer, to kernel-space. Examples include TriforceLinuxSyscallFuzzer [75], based on TriforceAFL [74], which enables full-system fuzzing through QEMU [12], and Oracle's similar QEMU-based kernel-fuzzing [69]. Leveraging Intel Processor Trace, kAFL [79] achieves precise, low-overhead kernel fuzzing. For coverage information, they rely on mechanisms such as QEMU's tracing functionality [75, 79] or kernel instrumentation (e.g., using `KCOV`) for coverage data [40].

Google's Syzkaller [40] is a state-of-the-art kernel fuzzer that essentially adopts a grammar-based approach to generate specialized test programs called *syz-programs*, which invoke target syscalls (see Figure 2). To automate their construction, Syzkaller uses *syz-descriptions*, written in a domain-specific language called *syzlang*. Syzlang expresses syscall dependencies using `resource` constraints. For example, `setsockopt()`, a socket syscall, requires a socket-type file descriptor (see Figure 1a). When generating a syz-program, if `setsockopt()` is selected, Syzkaller resolves its required resource by adding a preceding `socket()` call. The socket descriptor returned by `socket()` is then used as the first argument for `setsockopt()`.

Large Language Models (LLMs). Modern LLMs such as Claude [8], GPTs [68], and Gemini [42] excel at tasks ranging from natural language understanding to code generation [30, 51]. Given an appropriate prompt, they can synthesize high-quality code. For example, when asked to generate a C program for computing the inverse square root, ChatGPT outputs the classic implementation using the magic constant `0x5F3759DF`, rather than a naive calculation of $\frac{1}{\sqrt{x}}$.

3 Motivation and Challenges

Most socket-based syscalls are inherently stateful, meaning their behavior depends on both user-space program variables and kernel internal data structures. Effective fuzzing requires synthesizing the necessary context before invoking target syscalls and maintaining it for subsequent calls. Existing kernel fuzzers [16, 18, 19, 36, 40, 52, 70, 85, 86, 90, 98, 103, 105, 106] struggle with this due to four key challenges.

C1-*Abstract Wrappers*. Many POSIX socket-based syscalls are abstract wrapper functions (technically, C function pointers) that dispatch to submodule-specific implementations depending on arguments and kernel state. When a submodule is loaded, it registers its own implementations; for instance, the IPv4 TCP stack binds `inet_listen()` to `listen()` for TCP sockets, so `listen()` is dispatched to `inet_listen()` only if `sockfd` is created by `socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)` (see Figure 1b).

If a fuzzer invokes such wrappers with random arguments, the kernel often rejects them since no valid dispatch is possible, reducing fuzzing efficiency. Function prototypes alone, shown in Figure 1a, provide little guidance for generating valid calls.

To address this, Syzkaller allows test programs (syz-programs) to directly invoke submodule-specific implementations, e.g., `socket$inet_tcp()` instead of the generic `socket()` (see Figure 2b). While syz-descriptions simplify this process, they require explicitly specifying internal syscalls. Consequently, Syzkaller still depends heavily on human-written syz-descriptions or handcrafted syz-programs—a labor-intensive and error-prone task requiring expert knowledge.

C2-*Weak Typing*. Abstract wrappers often use weakly typed parameters to support multiple submodule implementations, making their interpretation highly context-dependent. For example, the fourth argument of `setsockopt()` (*i.e.*, `optval`) is declared as `void*`, but its meaning depends on the first three arguments (see Figure 1a). Setting `level = SOL_SOCKET` dispatches the call to `sock_setsockopt()`, whereas `level = SOL_TCP` dispatches to `tcp_setsockopt()`. Depending on the dispatched function, `optval` may range from a simple integer to a complex structure. For instance, Figure 1b shows `setsockopt()` configuring `SO_REUSEADDR`, where `optval` is an integer.

Randomly mutating arguments in such syscalls usually fails, since the relationships between parameters are non-trivial. To address this, Syzkaller introduces specialized variants with restricted argument types, such as `setsockopt$sock_int()` for integer socket options and `setsockopt$tcp()` for TCP options. It also requires explicit references in syz-programs or syz-descriptions, along with detailed definitions of argument structures (e.g., `optval`) to ensure correctness. Similar to **C**1, this approach still relies on expert-written syz-descriptions, making the process manual and error-prone.

C3-*Implicit Dependencies*. Many syscalls modify socket state inside the kernel in ways invisible at the syscall interface. For instance, enabling `SO_REUSEADDR` with `setsockopt()` changes how `bind()` behaves, yet this dependency is not reflected in their function signatures (see Figure 1a). Such hidden dependencies are crucial for exercising deeper syscall sequences but are hard for fuzzers to capture or generate.

Syzkaller models explicit dependencies via syzlang's `resource` constraints (e.g., a socket created by `socket()` can be reused in `setsockopt$sock_int()`), but it cannot express implicit ones, such as between `setsockopt$sock_int()` and `bind()`. As a result, Syzkaller may generate programs that omit `bind()` entirely (see Figure 2b), or include it only by chance, leaving many semantically meaningful paths unexplored.

Even when explicit dependencies exist, Syzkaller's candidate selection can still miss deeper call chains. For example, if syscall `C()` depends on `B()`, and `B()` depends on `A()`, Syzkaller can select only `A()` and `B()`, ignoring `C()`—even if `C()` is the syscall that finalizes the sequence and contributes most to the coverage. Such

```c
int socket(int domain, int type, int protocol);
int setsockopt(int sockfd, int level, int optname, const void *
    optval, socklen_t len);
int bind(int sockfd, const struct sockaddr *addr, socklen_t
    addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

int connect(int sockfd, const struct sockaddr *addr, socklen_t
    addrlen);
```

(a) System Call Prototypes

```c
int main() {
  int s, ns, buf[1024], enable = 1;
  if ((s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {exit
      (1); /* error */}
  if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(
      enable)) < 0) {exit(1);}
  ... /* load addr */
  if (bind(s, ... ) < 0) {exit(1); /* error */}
  if (listen(s, 1) < 0) {exit(1); /* error */}
  if ((ns = accept(s, ... )) < 0) {exit(1);}
  /* omitted ... now receive from new socket (ns) and take actions
      */
}
```

(b) A basic TCP server in C

```c
int main() {
  int s, status;
  if ((s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {exit(
      1);}
  /* addr is loaded with server's IP and PORT */
  if (connect(s, ...) < 0) {exit(1);}
  send(s, "HELLO", strlen("HELLO"), 0);
  /* omitted */
}
```

(c) A basic TCP client in C

Figure 1: Sample programs using socket system calls

```
// SOL_SOCKET = 1 // SO_REUSEADDR = 2
resource sock[fd]
sockopt_opt_sock_int = ..., SO_REUSEADDR, ...
socket$inet_tcp(domain const[AF_INET], type const[SOCK_STREAM],
    proto const[0]) sock
setsockopt$sock_int(fd sock, level const[SOL_SOCKET], optname
    flags[sockopt_opt_sock_int], optval ptr[in, int32], optlen
    len[optval])
bind(fd sock, addr ptr[in, sockaddr_storage], addrlen len[addr])
listen(fd sock, backlog int32)
...
```

(a) A syz-description

```
r0 = socket$inet_tcp(0x2, 0x1, 0x0)
setsockopt$sock_int(r0, 0x1, 0x2, &(0x7f00000001c0)=0x1, 0x4) //
    SO_REUSEADDR = 0x2
```

(b) A syz-program

Figure 2: Sample syz-description and syz-program

partial chains produce spurious or less effective test cases, reducing fuzzing efficiency.

C4-*Remote Endpoint Influence*. Networked programs fundamentally rely on exchanges between two or more endpoints. The outcome of a syscall often depends on how the remote endpoint responds, which may also update the socket's internal state in the kernel. For example, a TCP client's `send()` requires a *connected* socket, but if `connect()` fails due to a missing or uncooperative endpoint, subsequent `send()` calls will abort (see Figure 1c). Thus, meaningful exploration of deeper syscall sequences requires valid remote responses; without them, fuzzing can stall or derail.

Syzkaller sidesteps the need for real endpoints by emulating packet exchanges with special syscalls (*e.g.*, `syz_emit_ethernet()`) [38]. These syscalls inject packets into the kernel via a virtual NIC. However, this mechanism is primitive: injected packets often mismatch the current context (*e.g.*, a TCP packet during UDP fuzzing), and even Syzkaller's attempts to reuse extracted fields (like TCP sequence numbers) fall short of modeling realistic interactions [39]. Consequently, Syzkaller and fuzzers [16, 18, 19, 36, 52, 70, 85, 86, 90, 98, 103, 105, 106] built on its backend struggle to uncover deeper, semantically valid execution paths.

Limitations of Existing Fuzzers. Recent kernel fuzzers [9, 16, 18, 19, 36, 40, 45, 52, 70, 75, 79, 85, 86, 90, 98, 103, 105, 106] still face significant limitations against the challenges outlined above. Syzkaller, for example, can partially mitigate **C**1 and **C**2 with expert-written syz-descriptions, but this manual process is costly and scales poorly—covering only 38% of the Linux kernel source code [85]. To reduce this bottleneck, several works [18, 19, 70, 85, 98, 103, 105] explore automating input generation via seed mutation or syz-description synthesis. However, their effectiveness depends heavily on the quality and diversity of seed programs.

By contrast, **C**3 and **C**4 remain largely unresolved. Most fuzzers, including Syzkaller, fail to capture such dependencies. Some approaches [36, 45, 52, 70, 86] leverage trace analysis or static inspection to infer inter-syscall relations, but their precision is often low, leading to spurious or incomplete test cases.

A further drawback of Syzkaller-based fuzzers is their frequent generation of unreliable test programs that fail to reproduce bugs consistently. Not all crashing syz-programs or crash reports translate into reliable test cases. Our investigation of Syzkaller's continuous fuzzing platform, Syzbot [43], shows a reproduction success rate of only 61.5%. More than 70% of bug reports are ultimately ignored or discarded by the Linux kernel community because they cannot be reproduced reliably.

4 Our Approach: KnitFuzz

To address challenges **C**1–**C**4, we propose a novel end-to-end kernel fuzzer, KnitFuzz, for context-sensitive socket system calls.

4.1 Overview of KnitFuzz Design

Figure 3 shows the workflow of KnitFuzz, which consists of two decoupled stages: seed generation and kernel fuzzing. Given a target socket domain (*e.g.*, `AF_INET`, `AF_PACKET`, `AF_SMC`) and a set of system calls, the dataset generator produces combinations of randomly sampled calls. For each combination, the prompt generator creates program-generation tasks and feeds them to LLMs.

Candidate programs extracted from LLM responses are passed to verifiers, which ensure validity and compliance with requirements. If a program fails verification, an error message is generated, and both the program and error are fed back into the refinement loop until a preset iteration limit is reached. Programs that pass all checks are stored in a program vault and compiled into ELF binaries.

The kernel fuzzer, built on AFL++ [31], executes these binaries in a virtual machine and collects kernel coverage during fuzzing. To enable bi-directional network communication, KnitFuzz deploys a (dynamic) remote endpoint in the testbed. Importantly, the program generator and fuzzer operate independently, ensuring that program generation does not throttle fuzzing throughput.

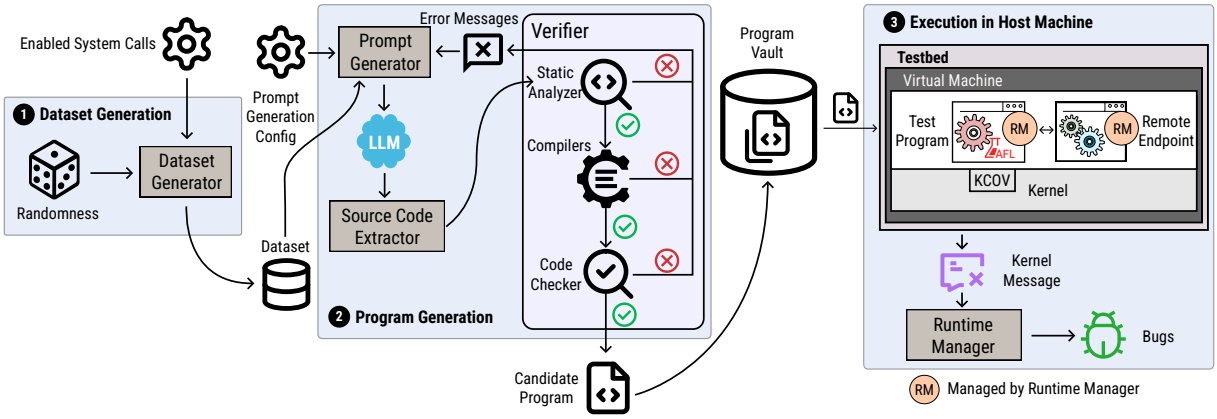


Figure 3: KnitFuzz Architecture

4.2 LLM-driven Seed Generation

Socket system calls are inherently context-sensitive, depending on both user-space data flow and hidden kernel state. Programs that incorrectly use syscalls or misuse parameters are often rejected by the kernel, limiting fuzzing effectiveness (see §3).

KnitFuzz addresses this by generating context-sensitive C programs that invoke socket system calls. C is the natural choice: it exposes syscalls directly, dominates LLM training data, and enables explicit verification of syscall usage. Bug-triggering C programs also serve as concrete artifacts, unlike Syzkaller's syz-programs, which are difficult to reproduce or translate in an actual test program. KnitFuzz leverages LLMs' prior exposure to syscall-invoking C code, eliminating the need for costly fine-tuning or elaborate in-context scaffolding.

The effectiveness of LLM-based code generation hinges on prompt design, but constructing prompts automatically is challenging. Socket-related syscalls are numerous (31 in total), and including all of them inflates context length and degrades output quality. Moreover, syscalls with parameter variability (*e.g.*, `setsockopt()`) expand the search space exponentially, making exhaustive prompt enumeration infeasible.

4.2.1 Candidate Programs and Verification. Given a target socket domain $\mathcal{D}$ and a set of system calls, KnitFuzz constructs a collection of syscall combinations $\mathbb{S}$ by sampling 5–9 calls at random, such that the total number of combinations exceeds the desired number of seed programs, $\mathbb{T}$ (*i.e.*, $|\mathbb{S}| > \mathbb{T}$). As shown in Algorithm 1, KnitFuzz queries LLMs iteratively until $\mathbb{T}$ valid socket programs are generated (see line 1.3).

At line 1.5, KnitFuzz invokes `GenerateAndVerify` (Function 2) for each element in $\mathbb{S}$, using a template prompt $\mathcal{G}$ and a set of key–value pairs $\mathcal{K}$. $\mathcal{G}$ specifies the task—*"Generate a socket program in C using socket system calls* `{syscalls}`"—where placeholders, like `{text}`, are dynamically substituted with values from $\mathcal{K}$ (see line 2.2).

With the constructed prompt, KnitFuzz queries the LLM to obtain a candidate program ψ (line 2.5). Since LLM outputs often contain errors, simple compilation is insufficient to diagnose issues, and programs may invoke third-party library calls instead of the requested syscalls. To address this, each ψ is checked with a chain of verifiers (line 2.6): (1) a *static analyzer* (*e.g.*, Clangd) for detailed diagnostics and code fixes, (2) a standard *compiler* (*e.g.*, GCC or Clang) to ensure successful compilation, and (3) a custom *syscall verifier* to confirm that all requested syscalls are present.

Algorithm 1: Seed Generation

Input : Syscall combinations $\mathbb{S}$; Socket domain $\mathcal{D}$; Seed size $\mathbb{T}$, where $|\mathbb{S}| > \mathbb{T}$
Output : A set $\mathbb{P}$, s.t. $\mathbb{P} = \{(P, \varphi) \mid P$ is a seed program and φ is a corresponding peer program$\}$. φ can be ϵ (none)

1.1 $\mathbb{P} \leftarrow \emptyset$ // Empty set
1.2 index $\leftarrow 0$
1.3 **while** $|\mathbb{P}| \neq \mathbb{T}$ **do** // $|\mathbb{P}|$ is the size of $\mathbb{P}$
1.4 funcs $\leftarrow \mathbb{S}$ [index++] // Extract a combination of syscalls from $\mathbb{S}$ and increment index
1.5 $P \leftarrow$ GenerateAndVerify ($\mathcal{G}$, {"syscalls": funcs}) // $\mathcal{G}$ is the prompt template for code generation
1.6 **if** $P \neq \epsilon$ **then** // A valid program generated
1.7 $P \leftarrow$ GenerateAndVerify ($\mathcal{M}$, {"syscalls": funcs, "program": P}) // $\mathcal{M}$ is the prompt template for instrumenting P to enable mutation
1.8 **if** $P \neq \epsilon$ **then**
1.9 **if** isPeerNeeded ($\mathcal{D}$) **then** // if a peer program needed
1.10 $\varphi \leftarrow$ GenerateAndVerify ($\mathcal{P}$, {"program": P}) // $\mathcal{P}$ is the template to generate a peer of P
1.11 **if** $\varphi \neq \epsilon$ **then**
1.12 $\mathbb{P} = \mathbb{P} \cup \{(P, \varphi)\}$
1.13 **else**
1.14 $\mathbb{P} = \mathbb{P} \cup \{(P, \epsilon)\}$
1.15 **return** $\mathbb{P}$

Function 2: GenerateAndVerify($\mathcal{T}$, $\mathcal{K}$)

Input : A prompt template $\mathcal{T}$, Key-value pairs $\mathcal{K}$
Output : program P

2.1 $P \leftarrow \epsilon$ // None
2.2 prompt $\leftarrow$ buildPrompt ($\mathcal{T}$, $\mathcal{K}$) // substitute each placeholder $\{k_i\}$ with corresponding v_i
2.3 $Funcs \leftarrow$ extractValueByKey ($\mathcal{K}$, "syscalls") // or None if empty
2.4 **for** $i \leftarrow 0$ **to** ρ **do** // ρ is our iteration limit for refinement
2.5 $\psi \leftarrow$ query_LLM (prompt) // ψ is a candidate program
2.6 isValid, error $\leftarrow$ checkWithEachVerifier (ψ, $Funcs$) // upon a failure, the chain of verifications stops
2.7 **if** isValid **then** // ψ has passed all verifiers
2.8 $P \leftarrow \psi$
2.9 **break**
2.10 prompt $\leftarrow$ buildPrompt ($\mathcal{R}$, {"prog": ψ, "err": error }) // prompt to fix **error** in program ψ using the refinement template $\mathcal{R}$
2.11 **return** P

If ψ passes all verifiers, it is returned to the caller (line 1.5) as a valid program. Otherwise, the verification process halts upon failure, and KnitFuzz constructs a new refinement prompt using $\mathcal{R}$ with updated placeholder values (line 2.10). To avoid infinite refinement, the process is bounded by a fixed iteration limit ρ, chosen empirically based on when further refinements no longer provide substantial improvements.

4.2.2 Mutations. Kernel fuzzing requires millions of program variants with abnormal or adversarial syscall arguments. While the method in § 4.2.1 can generate thousands of high-quality seeds, scaling seed generation directly with LLMs is cost-prohibitive. Asking LLMs to mutate programs online is also infeasible: precautionary defenses block malicious code generation [20, 93, 104], and coupling fuzzing with LLM queries would severely degrade throughput.

Executing each seed only once further underutilizes its potential. Mutation-based fuzzers [14, 91, 101] address this by producing vast numbers of low-cost variants—every simple mutation yields a new test case. However, naïve mutations risk breaking socket program syntax or semantics, producing invalid tests. Syzkaller-based fuzzers [18, 19, 36, 52, 70, 85, 86, 90, 98, 103, 105, 106] mitigate this by synthesizing *syz-programs* (Figure 2), but they remain limited in resolving implicit dependencies among socket syscalls—dependencies that are essential for producing valid, executable programs.

KnitFuzz instead mutates seed programs at the syscall interface. Rather than hardcoding mutated data—which would require source edits for each new variant and risk breaking the code—KnitFuzz adopts a reusable pathway that ensures fuzzed data flows directly into syscall arguments. We delegate selection of memory regions to instrument and preservation of structural and semantic relationships to LLMs. KnitFuzz uses a prompt template $\mathcal{M}$ (Figure 4a) and asks LLM to instrument seeds with a hook function, `fuzz(...)`. Inspired by LibFuzzer's `LLVMFuzzerTestOneInput()` [60], the hook enables each execution to consume fresh fuzzed data without recompilation, expanding a single seed into thousands of variants. For example, in an instrumented program, the buffer passed to `send()` is loaded with fuzzed data due to the preceding call to `fuzz(buffer, ...)` (Figure 4b). For multi-field structs, LLMs may either overwrite the entire struct or preserve specific fields (e.g., pointers). In the latter case, it records critical values before invoking 'fuzz()' and restores them afterward, maintaining semantic constraints.

4.2.3 Remote Endpoint. Test programs for several socket domains (*e.g.*, `AF_INET`, `AF_SMC`, `AF_UNIX`) cannot establish network communication on their own, necessitating a remote endpoint. Without an appropriate endpoint, fuzzing exploration can stall or fail (see **C**4 in §3). The endpoint must match the `domain` argument of `socket()`, which dictates the protocol family (*e.g.*, INET, SMC, UNIX).

A remote endpoint does not need to implement full peer functionality. Instead, a minimal peer that responds sufficiently to emulate a live endpoint is often adequate. However, responses must still conform to the target protocol, meaning an endpoint designed for one domain (*e.g.*, INET) cannot be reused for another (*e.g.*, SMC). As a result, constructing domain-specific endpoints is essential but can be challenging, often requiring expert knowledge or specialized hardware (*e.g.*, SmartNICs for RDMA).

For INET (`IPv4`) and INET6 (`IPv6`), KnitFuzz provides a reusable, lightweight endpoint that can dynamically operate as either a

```
Role: You are a Linux socket programmer writing programs in C. [...]
Context:
- {program}
- Assumes a header "fuzz.h" is available providing "void fuzz(void *
      buffer, int size);". This fuction will fill a buffer with fuzz
      data.
Task: Fill the memory locations (aka, buffers) that are going to be
      passed to syscalls in the given program using 'fuzz()'
```

(a) A conceptual prompt template to instrument a socket program

```
#include "fuzz.h" // for fuzz()
int main() {
  /* omitted */
  fuzz(buffer, sizeof(buffer));
  if (send(sockfd, buffer,
       sizeof(buffer), 0) == -1)
       /* omitted */
}
```

(b) The instrumented socket program

```
/* the "fuzz.h" implementing fuzz()
   */
inline void fuzz(void* buffer, int
      size){
  memset(buffer, 0, size);
  int buf_len = read(STDIN_FILENO,
       buffer, size);
  if (buf_len < 0) { perror("read(
       stdin)"); }
}
```

(c) A simplified `fuzz()` implementation

Figure 4: Injection of mutated data to syscall arguments

dummy server or client. The endpoint requires minimal development effort, as it is based on common protocol knowledge (*e.g.*, textbook state machines) to respond meaningfully according to IP/TCP/UDP semantics and built using mature third-party networking libraries. It runs inside the same VM as the test program and is monitored during fuzzing, pushing the test program along the transition in protocols' finite state machines.

For other domains, such as `AF_UNIX` and `AF_SMC`, KnitFuzz leverages LLMs to automatically generate a peer program compatible with the test program (see line 1.10 of Algorithm 1). The LLM is prompted to *"Generate a program that can communicate as the peer of the given* `{program}` *"* with constraints that it must not terminate and must handle multiple connections. The resulting peer program (φ) acts as the remote endpoint during fuzzing.

Finally, KnitFuzz pairs each instrumented seed program (P) with its corresponding peer (φ) and adds the pair to $\mathbb{P}$ (line 1.12 of Algorithm 1), iterating until $\mathbb{T}$ seed pairs are generated.

4.3 Coverage-Guided Fuzzing

KnitFuzz produces realistic C programs as seeds. However, Syzkaller's infrastructure cannot be reused, since it accepts only syz-programs rather than Linux executables. To address this, KnitFuzz introduces a custom fuzzing infrastructure.

Testbed. KnitFuzz employs a QEMU-based virtual machine (VM) [12] running the Linux kernel under test and repurposes AFL++ [31], a widely used coverage-guided fuzzer for user-space programs. AFL++ is well-suited for our design: it operates entirely in user space without kernel modifications, accepts fuzzed inputs via standard channels such as `STDIN` (aligning with our hook-based design in §4.2.2), and provides a rich set of mutators for strings, integers, and binary blobs that we reuse for mutating system call arguments. Since our focus is kernel rather than user-space coverage, test programs can be compiled with a standard compiler (*e.g.*, `clang`) instead of AFL-specific compilers (*e.g.*, `afl-clang-fast`).

By default, AFL++ fuzzes a single binary per run and relaunches for each new program. Our design follows the same principle: each AFL++ instance targets one test program at a time. However, AFL++ cannot be used as a drop-in solution for KnitFuzz, as two challenges arise. First, effective fuzzing requires attributing kernel coverage

exclusively to the currently executed test program, denoted P_i, and using that coverage to guide AFL++'s mutations. While we would like to execute P_i repeatedly and preserve AFL++'s mutation state across executions, allowing AFL++ to directly execute and fuzz P_i would reset its internal state, thereby discarding accumulated knowledge from prior mutations. Second, different test programs embed `fuzz()` hooks at different locations and require varying amounts of input. As a result, AFL++'s internal state from one program's (P_i) fuzzing campaign cannot be effectively transferred to another (P_j).

To address the first challenge, we develop a *fuzz-adapter*, inspired by [21]. AFL++ treats the fuzz-adapter as its target. During execution, AFL++ forks and runs the fuzz-adapter as a child process. The fuzz-adapter then forks and enabling KCOV for only the current process, excluding coverage from remote peer and asynchronous kernel activities. After that it executes a test program P_i via `execvp()`, which consumes AFL++ mutated inputs via `STDIN`. After P_i terminates, the fuzz-adapter collects kernel coverage for P_i using Linux's native `KCOV` module [58], replaces AFL++'s default coverage feedback with `KCOV` coverage, flushes the input buffer, and starts the next iteration, running P_i again. This design ensures that AFL++'s mutations are guided by kernel-level rather than user-space coverage, while allowing the fuzz-adapter itself to remain alive across multiple executions of P_i, thereby preserving AFL++'s view of a continuously running target.

To address the second challenge, we isolate fuzzing campaigns across test programs by rebooting the VM and restarting AFL++ whenever KnitFuzz switches to a new test program P_j.

Scheduler. KnitFuzz employs a *fuzzing scheduler* to orchestrate execution: it determines which program to run, how long to fuzz it, and when to switch to the next. Since generated programs are relatively simple, they typically reach a coverage plateau quickly. Inspired by Syzkaller's timeout strategy, KnitFuzz rotates programs once no new coverage is observed for a period of time. Conversely, programs showing promising coverage growth are granted extended fuzzing time to maximize exploration.

5 Implementation

We now discuss key implementation choices we made for KnitFuzz.

Prompt Generator. The prompt generator, implemented in Python 3.12 (282 LoC), constructs prompts based on predefined templates, syscall combinations, and the target domain. These prompts instruct the LLM to generate test programs with fuzzing entrypoints and, when required, corresponding peer programs.

Source Code Extractor. Although prompts request only source code, some LLMs return additional comments or explanations. To handle this, we implemented a lightweight extractor in Python 3.12 (7 LoC) that parses markdown code fences and isolates the generated source code.

Verifiers. The syscall verifier, implemented as a compiler pass on LLVM 13 (154 LoC in C++), checks generated programs to ensure that the required system calls are present. We also utilize clangd, GCC, and Clang to make sure testcases are compilable and have no shallow bugs.

LLM Gateway Server. KnitFuzz relies on LLMs solely for inference, but since each model exposes its own interface, a unifying API is necessary. To this end, we implemented a server that provides an HTTP interface, abstracting away model-specific details and ensuring portability and flexibility in deployment. By leveraging this interface, KnitFuzz remains LLM-agnostic.

Fuzz-Adapter. The fuzz-adapter, implemented in C (1,151 LoC), acts as a mediator between AFL++ and the test program inside the VM. A child process may hang, potentially blocking the fuzz-adapter and AFL++. To prevent wasted cycles, KnitFuzz incorporates a *runtime monitor* within the VM. Running alongside AFL++, the adapter, and test programs, the monitor detects unresponsive test programs and notifies the scheduler to recover or rotate programs.

During fuzzing, the adapter replaces AFL++'s coverage feedback with the syscall coverage collected by enabling `KCOV` before executing the test program via `execvp()`. Beyond bridging AFL++ with the kernel, it also manages the network namespace and triggers necessary network interactions. Additionally, the adapter continuously monitors kernel messages: upon detecting non-fatal exceptions (*e.g.*, `oops`), it briefly sleeps, allowing the external runtime manager to collect runtime information such as AFL++ fuzz inputs. Since the manager checks more frequently than the sleep interval, this design avoids communication overhead between the host and VM.

Runtime Manager. The runtime manager, implemented in Python 3.12 (1,202 LoC) without third-party libraries. It spawns AFL++ and remote endpoint processes inside each VM, then periodically collects runtime information and monitors for crashes. When a crash occurs, the manager leverages the QMP protocol to restore the VM state, avoiding the overhead of a full reboot.

The runtime manager also supports two levels of parallelism: *Instance-level* enables multiple KnitFuzz instances to run on the same machine without interference, while *VM-level* allows multiple kernels to be fuzzed and monitored simultaneously.

Remote Endpoint. The remote INET endpoint is implemented in Go using third-party libraries, with 190 LoC for TCP handling, 14 LoC for UDP logic, and 108 LoC for IP processing, all derived from textbook protocol knowledge. Instead of using the loopback interface (`127.0.0.1`), which bypasses key kernel paths, KnitFuzz employs a virtual `tun` device [59]. The socket program interacts with one end of the device as a regular network interface, while the endpoint communicates with the other end via file I/O. The endpoint remains lightweight by avoiding per-connection state and operates within a dedicated Linux network namespace to capture all packets, including those with mutated destination addresses. To conserve testbed resources, it enforces flow control by discarding packets once a predefined cap is reached.

For domains such as `AF_UNIX` and `AF_SMC`, KnitFuzz leverages LLMs to generate peer programs, requiring only 5 lines for the prompt and 10 lines for environment setup. During fuzzing, the peer corresponding to the test program is launched as a remote endpoint using `systemd-run` with the `restart always` option. For SMC, an `rxe` device (Soft-RoCE) is additionally configured to enable RDMA connections.

6 Evaluation

Research Questions. We evaluate KnitFuzz through the following research questions: **(RQ1)** What configuration yields the most effective seed programs? (§ 6.1) **(RQ2)** How does KnitFuzz compare to

other kernel fuzzers in terms of code coverage? (§ 6.2) **(RQ3)** How does KnitFuzz perform on other socket domains in terms of code coverage? (§ 6.3) **(RQ4)** How effective is KnitFuzz at discovering bugs relative to other fuzzers? (§ 6.4)

Dataset. Our dataset consists of POSIX socket-related system calls extracted from the `proto_ops` C structure in *include/linux/net.h*, along with `epoll`-related system calls, totaling 31 syscalls.

Experimental Setup. We evaluated KnitFuzz with 3 cost-effective LLMs, including OpenAI GPT 5 mini, Anthropic Claude 4 Sonnet, and Google Gemini 2.5 Flash Lite. All models were accessed via their API endpoints with default hyperparameters. To avoid bias, each program was generated in a fresh inference session without chat history. Unless otherwise noted, Gemini 2.5 Flash Lite was used for the evaluation of RQ2–RQ4.

Experiments were conducted on two identical machines, each with an AMD EPYC 7702P CPU (64 cores, 128 threads, 256 MB cache, 2.00 GHz) and 475 GB RAM. Each testbed ran inside a VM with 2 CPUs and 2 GB memory. For each KnitFuzz instance, 16 CPUs were allocated, enabling 8 testbeds in parallel and thus up to 8 instances of KnitFuzz per machine.

Coverage Collection. Kernel coverage of generated programs on Linux v6.12.42 was measured using the fuzz-adapter leveraging the native `KCOV` module. Each program was allocated 15 minutes of fuzzing, extended by 3 minutes if new blocks were covered, and skipped after 10 minutes of stagnation. Campaigns ran for 24 hours, with results averaged over ten runs [53, 78]. Total *block* and *branch coverage* were computed as the deduplicated sum across programs.

6.1 Seed Generation

To generate high-quality seeds, we first determined the optimal selection of key parameters of KnitFuzz through systematic empirical experiments. All LLM-specific hyperparameters are kept at their platform defaults.

Refinement Iteration (ρ). LLM-generated seeds can be invalid, so KnitFuzz employs an iterative refinement process with a maximum of ρ iterations. For this experiment, we used GPT 5 mini to generate seeds. To determine an effective limit, we varied the number of refinement attempts and measured the *success rate*—the proportion of valid programs generated out of the total number of attempts. Experiments with up to 10 iterations (Table 1) showed diminishing returns: improvements plateaued beyond 3 iterations, while computational costs continued to rise. Therefore, setting $\rho = 3$ provides an effective cutoff.

Effective LLM. To identify the most effective LLM for seed generation among 3 candidates, we evaluated each model based on the code coverage of generated seed programs. Figure 5 show that Gemini 2.5 Flash Lite outperformed all other models, achieving 1.67% and 7.56% higher coverage than Claude 4 Sonnet and GPT 5 mini, respectively (for basic block coverage, see Figure A1). Additionally, Gemini also offered the lowest cost per million tokens.

Seed Size ($\mathbb{T}$) and Diversity. KnitFuzz decouples seed generation from fuzzing execution, thereby avoiding any impact on fuzzing throughput. A potential risk, however, is that the system may exhaust available seeds during a campaign. To mitigate this, we provisioned $\mathbb{T} = 300$ per domain, totaling 1,500 across all five domains. Given the previously defined per-program time limit of 15 minutes (L), with dynamic extension if shown improved coverage, KnitFuzz can fuzz at most 768 seeds within a 24-hour campaign. This upper bound remains well below the generated pool, ensuring that seed availability does not constrain fuzzing.

Table 1: Determine iteration limit for refinement

Max. Iter.	Succ. Rate	Token (I/O)	Total Cost (USD.)
0	61.9%	5569/6102	13.60
1	92.8%	7899/7680	17.33
2	99.1%	8717/8175	18.53
3	99.8%	8883/8268	18.76
4	100%	8924/8292	18.81
5	100%	8924/8292	18.81
≈≈ *remaining rows are omitted* ≈≈			

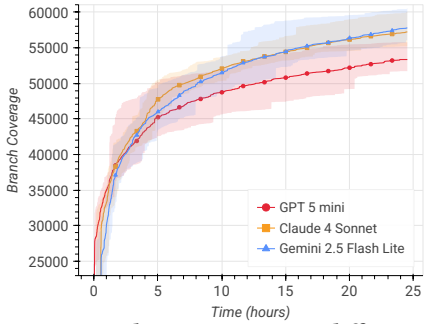


Figure 5: Branch coverage across different LLMs

To quantify the diversity of LLM-generated seeds, we computed pairwise similarity across the 1,500 seed pool using LLVM's IRSimilarity at the IR level. On average, each program exhibited only 8.4% similarity to other programs in the pool (min 0.8%, max 34.6%), confirming that the generated seeds are highly diverse and unlikely to exercise on the same code paths. We further sampled 50 files from the generated C programs; 84% contained implicit dependencies correctly resolved by LLMs.

6.2 Comparison with Other Fuzzers

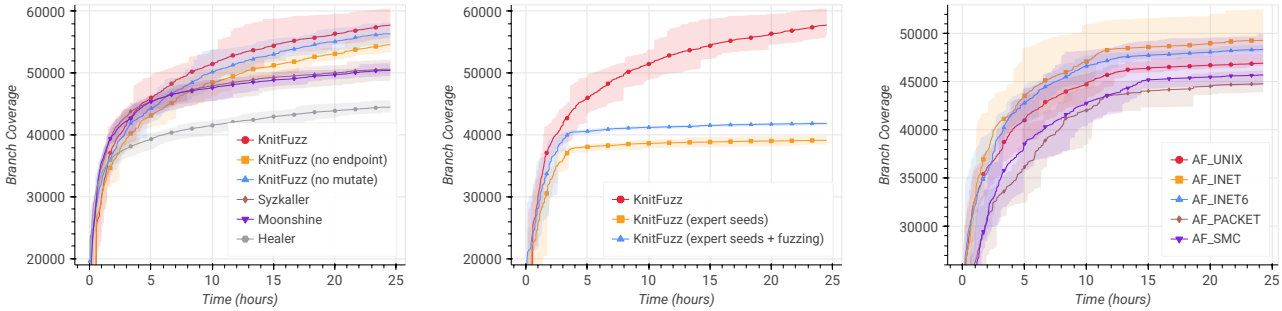
To evaluate KnitFuzz's effectiveness, we compared it against three state-of-the-art kernel fuzzers: Syzkaller [40], Healer [86], and Moonshine [70]. We also attempted to port Actor [36], kAFL [79], FuzzNG [16], and SyzDirect [88], but were unsuccessful. For instance, FuzzNG and kAFL require Intel VT-x and Intel PT, respectively, which were unavailable on our AMD-based platform. SyzDirect depends on a pre-selected bug dataset, but the public release contains errors (See Appendix D for details).

For fairness, all fuzzers were restricted to networking-related syscalls drawn from our dataset described in §6. Modifications to baseline tools were minimal, incurring only negligible one-time startup overhead.

We further ablated KnitFuzz by disabling three components individually, yielding variants "KnitFuzz (expert seeds)", "KnitFuzz (expert seeds + fuzzing)" "KnitFuzz (no mutate)" and "KnitFuzz (no endpoint)." This allowed us to isolate the contribution of LLM-generated seeds, mutation, and remote endpoint support.

We use branch coverage as the main evaluation metric. Basic block coverage is also reported in Appendix A. Each fuzzer ran ten independent 24-hour campaigns, and results were aggregated.

Figure 6a reports branch coverage, with solid lines showing medians and shaded regions indicating the full range (min~max) across runs. KnitFuzz outperformed all competing fuzzers, with an



(a) KnitFuzz vs. other fuzzers (all domains) **(b) KnitFuzz vs. expert seeds** **(c) KnitFuzz on different domains**

Figure 6: Performance of KnitFuzz in terms of code (branch) coverage

average gain of 19.52%. Tables A1 and A2 present the improved coverage of KnitFuzz over all other fuzzers.

In Figure 6b, "KnitFuzz (expert seeds)", "KnitFuzz (expert seeds + fuzzing)" utilize an expert-seed pool by collecting 90 real-world C programs from three datasets: Linux Testing Project, kselftests, and Glibc testsuite. After 24 hours without mutation, this pool reached a maximum of 39900 branch coverage. After enabling mutation with `fuzz()`, it reached 42064. KnitFuzz achieved a minimum of 56,045 across 10 runs, substantially outperforming this pool and demonstrating the effectiveness of LLM-generated seeds.

Moreover, "KnitFuzz (no mutate)" and "KnitFuzz (no endpoint)", reveal that beyond the LLM-generated seeds, the remote endpoint module contributed more to coverage gains than mutation. This underscores the importance of addressing **C**4 in kernel fuzzing. Although mutation added less to coverage, it remains essential for bug discovery (§ 6.4): KnitFuzz detected one bug without the endpoint but none without mutation. Finally, both ablated versions outperformed prior fuzzers, primarily due to LLM-generated seeds.
Takeaway. *KnitFuzz outperforms prior fuzzers with average gains of 19.52% in branch coverage.*

6.3 Performance on Different Socket Domains

We evaluated KnitFuzz on each of the 5 target domains individually. We used custom-built remote endpoints for `AF_INET` and `AF_INET6`, LLM-generated peer programs for `AF_UNIX` and `AF_SMC`, and none for `AF_PACKET`. Figure 6c shows the resulting branch coverage (see Figure A3 for basic block coverage). `AF_INET` and `AF_INET6` achieved the highest coverage across both metrics.

6.4 Bug Discovery

To compare KnitFuzz's vulnerability detection capability with other fuzzers, we fuzzed five socket-related subsystems in Linux v6.12.42 (LTS), focusing on protocol domains such as `AF_INET`, `AF_INET6`, `AF_UNIX`, `AF_PACKET`, and `AF_SMC` (Shared Memory Communication over RDMA). Identical compute resources and time budgets were allocated for all fuzzers.

Table B3 summarizes the discovered bugs and their current status. KnitFuzz found two bugs in the `AF_SMC` subsystem, whereas other fuzzers detected only one (Bug2) and no additional issues. Moreover, KnitFuzz was the only fuzzer that produced C test programs (PoCs) for these bugs, while others failed to produce any.

Bug1: Deadlock in `smc_listen_work` Cancellation. KnitFuzz uniquely discovered a race condition between `smc_clcsock_release()` and `smc_listen_work()`. When `close()` cancels an in-progress `listen()` worker, the lock held by the worker is never released, leaving `close()` blocked during cleanup and rendering the kernel unresponsive. This bug was not detected by any other fuzzer and has since been reported (For PoC, see Appendix C).
Bug2: Null Pointer Dereference in `smc_tcp_syn_recv_sock()`. KnitFuzz exposed a null pointer dereference in `net/smc/af_smc.c`, detected via ASAN. As shown in Figure B4, the function at Line 2 may return `NULL`, yet subsequent code dereferences the pointer (*i.e.*, `smc`), triggering a fatal kernel exception. Other fuzzers also triggered this bug, but their reports lacked reproducible test cases.
Takeaway: *KnitFuzz outperforms related fuzzers, detecting one previously unknown bug and providing reproducible C test programs for both bugs, aiding kernel developers in debugging.*

7 Discussion

While KnitFuzz achieves strong performance by addressing the challenges outlined in § 3, these benefits come with trade-offs.
Limitations of LLMs. KnitFuzz is most effective at solving **C**1–**C**3 for various socket programs. This is because LLMs tend to generate only the most common usage patterns unless carefully guided [76]. Extending KnitFuzz to cover more diverse use cases requires expanding the syscall dataset, which can be automated by extracting entries from structured system call tables [15]. However, many advanced syscall variants remain undocumented in kernel sources [89]. If an advanced usage is unknown to us, it is unlikely that LLMs can generate it from training data; in such cases, exploration relies heavily on the mutation module.
High Coverage but Less Bugs. Although KnitFuzz achieved higher coverage, it discovered only one new and one known bug in the `AF_SMC` subsystem. Many other subsystems have existed for decades and have undergone extensive bug-finding efforts, with earlier IPv4-related bugs already fixed in Linux v6.12. Prior Syzkaller-based fuzzers that reported numerous bugs targeted the entire kernel rather than specific networking subsystems like ours. Nonetheless, KnitFuzz offers a clear advantage by generating a valid proof-of-concept for every bug it discovers, unlike previous approaches.
Remote Endpoints. While a domain-specific remote endpoint improves coverage, KnitFuzz supports both custom-built and LLM-generated endpoints with minimal development effort (see § 5). We acknowledge that remote endpoints are socket-domain specific and

not directly reusable. However, basic endpoints can be constructed using common protocol knowledge (e.g., TCP state machines), and the same approach can be adapted to other protocols, such as QUIC using two `io.Copy` operations between a raw socket client and a `quic-go` server. For more complex protocols, LLM-generated peers can serve as drop-in replacements.

Comparison with expert-level seeds or syz-programs. To the best of our knowledge, no benchmark of C-based socket programs exists, preventing a comparison between LLM-generated seeds and expert-crafted C-seeds. Existing protocol-fuzzing repositories, such as ProfuzzBench [67], focus on application-layer protocols (e.g., HTTP, SSH), shifting attention from kernel networking subsystems to application logic. Similarly, Syzkaller-based syz-programs cannot be used directly in KnitFuzz without substantial engineering effort.

Seed Generation Latency and Cost. Seed generation latency is hard to measure because LLM APIs are rate-limited and vary with server load. Table 1 shows per-program token usage; after three refinements, it takes on average 8883/8268 (I/O) tokens. Total costs for the entire seed set using GPT 5 mini, which are minimal.

Threats to Validity. As with any empirical study, our results are subject to validity threats. A threat to external validity concerns the generalizability of our findings. We selected LLMs from three major vendors, but KnitFuzz depends on vendor-specific inference APIs. Since both model internals and APIs evolve over time, future changes may affect KnitFuzz's performance. A threat to internal validity lies in the correctness of our implementation. While KnitFuzz may itself contain bugs, we have released our implementation and evaluation scripts to enable scrutiny.

8 Related Work

We now present some additional closely related prior research.

Network Protocol Fuzzing. A wide range of network protocol fuzzers have been developed to explore vulnerabilities in stateful communication protocols. Protocol state fuzzers [13, 23, 32–35, 48, 49] either infer or rely on user-provided state machines to guide fuzzing by sending crafted protocol-specific message sequences and observing responses. Mutations are typically applied at the message level (e.g., altering sequence or delivery), with inputs drawn from prior sessions or user specifications.

Black-box fuzzers such as Peach, BooFuzz, Sulley, SPIKE, and SNOOZE [1–3, 5, 11] can fuzz stateful protocols but require detailed protocol models or user-supplied scripts, making them labor-intensive and less reliable. For instance, Peach relies on protocol-specific XML grammars, and BooFuzz requires about 100 lines of Python code per protocol describing packet formats and communication orchestration. In contrast, KnitFuzz requires minimal effort to incorporate a remote endpoint, by leveraging either textbook state machines or LLM-generated peers (see § 5).

Stateful coverage-guided fuzzers [7, 10, 34, 54, 64, 66, 73, 80] extend traditional fuzzing with token-level and message-level mutations, yet still lack context-sensitive handling of protocol semantics. However, none of these fuzzers are designed to fuzz the Linux kernel, as they primarily assume a remote attacker sending crafted protocol messages rather than invoking system calls through user-space programs.

Specific efforts on TCP fuzzing [48, 49, 72, 107] have explored message-level input generation. Among them, TCP-Fuzz [107] introduced a two-dimensional approach that considers both syscalls and messages when constructing test case sequences for TCP implementations. However, its test cases are generated based on some predefined TCP-specific rules, limiting their applicability to other parts of the TCP/IP (INET) stack.

Kernel Fuzzing. Kernel fuzzers instead generate programs invoking system calls. To preserve context-sensitivity, prior work has explored manual specifications (Syzkaller [40]), automated specification inference (KSG [85], SyzGen [19], SyzGen++ [18], KernelGPT [98], SyzGPT [105]), seed corpus construction (Moonshine [70], ECG [103]), static analysis (HFL [52]), dependency inference (Healer [86], ACTOR [36], SyzVegas [90], StateFuzz [106]), and rule-based fuzzers (TriforceAFL [75], kAFL [79], Trinity [9], FuzzNG [16], IMF [45]). Despite these advances, existing approaches remain limited in resolving implicit dependencies, particularly among socket-related system calls, which are critical for producing valid and executable programs.

Two main strategies exist for driving kernel fuzzing: on-device fuzzing and emulation-based fuzzing. While on-device approaches [4, 17, 29, 55, 83] are valuable for exposing hardware-specific defects, they suffer from limited control, higher setup cost, and fragile crash handling. In contrast, emulation-based fuzzing [16, 40, 61, 71, 74, 77] offers a scalable and cost-effective alternative by running kernels in virtualized environments such as QEMU, which enables complete monitoring, deterministic replay, and flexible introspection. Coverage collection is another crucial component of fuzzing effectiveness. Although binary rewriting or non-invasive tracing methods are useful when source code is unavailable [4, 17, 27, 65, 83], they often introduce overhead or restrict feedback fidelity. In comparison, source-based instrumentation through `KCOV` provides lightweight and precise coverage feedback, making it particularly well-suited for syscall-driven kernel fuzzing [40, 70, 86, 105].

On-device fuzzers often rely on dedicated peripherals [61, 83] or network interfaces [4, 17] to fuzz the target kernel or drivers. In contrast, emulation-based fuzzers [16, 36, 40, 52, 70, 85, 86, 98, 103, 105] do not directly use remote endpoints. Instead, they emulate packet delivery from a remote endpoint by embedding domain-specific functions into test programs. However, such emulations are typically uncontrolled, often generating responses that are irrelevant to the target protocol. This misalignment derails deeper path exploration and reduces fuzzing efficiency.

LLM-driven Test Generation. LLMs have been applied to generate diverse test inputs, including programs for deep learning libraries [25, 26], compilers [94, 97, 100], and SMT solvers [87]. They have also been used to synthesize fuzz drivers [102], mutate seeds [47], and generate test cases or seed examples from natural-language requirements [6, 63]. More recently, LLMs have been employed for syz-program generation in kernel fuzzing [98, 105].

However, pre-trained LLMs are often unreliable for domain-specific inputs. Prior work addressed this via fine-tuning on bug-triggering corpora [24], in-context learning with few-shot examples [25, 64, 98, 102, 105], or full re-training [87].

In contrast, no prior effort targets generating user-space C programs for fuzzing the kernel's socket interface. Our advantage lies

in leveraging LLMs' existing exposure to syscall-invoking C code, eliminating the need for costly fine-tuning or in-context scaffolding.

Program Synthesis. Automated synthesis of syscall-based C programs faces both theoretical and practical roadblocks. Theoretically, synthesis reduces to second-order logic, which is undecidable in general [22]. Practically, search-based frameworks like CEGIS collapse under search-space explosion without highly restrictive templates [50]. Even producing syntactically valid but semantically meaningless C programs requires major engineering effort just to avoid undefined behavior [99], or demands heavyweight techniques which are limited to narrow domains [46]. Given additional challenges such as pointer aliasing, nondeterminism, and the idiosyncrasies of syscalls, synthesizing correct syscall-driven C programs remains infeasible.

9 Conclusion

We introduced KnitFuzz, which leverages LLMs' semantic understanding to address the context sensitivity of socket system calls and integrates a coverage-guided userspace fuzzer to enable argument-level mutations, thereby improving fuzzing efficiency. Our evaluation shows that, compared to state-of-the-art kernel fuzzers—Syzkaller [40], Moonshine [70], and Healer [86]—KnitFuzz improves coverage by 14.12%, 14.59%, and 29.84%, respectively. Moreover, KnitFuzz uncovered two kernel bugs, underscoring its practical effectiveness in uncovering real-world vulnerabilities.

Acknowledgments

This research was supported by the NSF award CNS-2339350.

References

[1] 2011. Peach Fuzzer: Discover unknown vulnerabilities. https://gitlab.com/peachtech/peach-fuzzer-community. accessed Jul 2022.
[2] 2012. boofuzz: Network Protocol Fuzzing for Humans. https://github.com/jtpereyda/boofuzz. accessed Jul 2022.
[3] 2014. OPENRCE. Sulley: A pure-python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley.
[4] 2019. LLDBFuzzer: Debugging and fuzzing the apple kernel. https://www.trendmicro.com/en_us/research/19/h/lldbfuzzer-debugging-and-fuzzing-the-apple-kernel-with-lldb-script.html.
[5] 2019. SPIKE. https://www.kali.org/tools/spike/. accessed Jul 2022.
[6] Joshua Ackerman and George Cybenko. 2023. Large language models for fuzzing parsers (registered report). In *ISSTA FUZZING 2023*.
[7] Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: High-Throughput Fuzzing of Network Applications. In *ISSTA*. doi:10.1145/3533767.3534376
[8] Anthropic. 2023. Claude. https://claude.ai/.
[9] Trinity Authors. 2012. trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity.
[10] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful greybox fuzzing. In *USENIX Security 22*. 3255–3272.
[11] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. 2006. SNOOZE: Toward a stateful network protocol fuzzer. In *ISC*. Springer.
[12] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX ATC*, Vol. 41. 10–5555.
[13] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2015. A messy state of the union: Taming the composite state machines of TLS. In *IEEE S&P*. 535–552.
[14] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.
[15] Marco Bonelli. 2025. Linux kernel syscall tables. https://syscalls.mebeim.net/?table=x86/64/x64/v6.10.
[16] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. 2023. No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions. In *NDSS*.
[17] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *NDSS*.
[18] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Schales, Jiyong Jang, and Zhiyun Qian. 2024. SyzGen++: Dependency Inference for Augmenting Kernel Driver Fuzzing. In *IEEE S&P*.
[19] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. 2021. Syzgen: Automated generation of syscall specification of closed-source macos drivers. In *ACM CCS*. 749–763.
[20] Zixuan Chen, Weikai Lu, Xin Lin, and Ziqian Zeng. 2025. SDD: Self-Degraded Defense against Malicious Fine-tuning. *arXiv preprint arXiv:2507.21182* (2025).
[21] Cloudflare. 2019. A gentle introduction to Linux Kernel fuzzing. https://blog.cloudflare.com/a-gentle-introduction-to-linux-kernel-fuzzing/.
[22] Cristina David and Daniel Kroening. 2017. Program synthesis: challenges and opportunities. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017), 20150403.
[23] Joeri de Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *USENIX Security*.
[24] Yao Deng, Zhi Tu, Jiaohong Yao, Mengshi Zhang, Tianyi Zhang, and Xi Zheng. 2025. Target: Traffic rule-based test generation for autonomous driving systems. *IEEE Transactions on Software Engineering* (2025).
[25] Y. Deng, C. Xia, C. Yang, S. Zhang, S. Yang, and L. Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *ICSE*.
[26] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *ISSTA*.
[27] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *IEEE S&P*. IEEE, 1497–1511.
[28] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. In *MSR*. IEEE, 131–142.
[29] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. 2023. Fuzzing embedded systems using debug interfaces. In *ISSTA*. 1031–1042.
[30] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, and Daxin Jiang. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv:2002.08155* (2020).
[31] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *USENIX WOOT 20*.
[32] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *CAV*. Springer International Publishing.
[33] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri De Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of DTLS implementations using protocol state fuzzing. In *USENIX Security 20*. 2523–2540.
[34] Paul Fiterău-Broştean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2022. Dtls-fuzzer: A dtls protocol state fuzzer. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 456–458.
[35] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. 2017. Model Learning and Model Checking of SSH Implementations. In *Proc. of SPIN*. ACM.
[36] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. 2023. ACTOR:Action-Guided Kernel Fuzzing. In *USENIX Security 23*. 5003–5020.
[37] Google. 2013. AddressSanitizer, ThreadSanitizer, MemorySanitizer. https://github.com/google/sanitizers.
[38] Google. 2015. External network fuzzing for Linux kernel. https://github.com/google/syzkaller/blob/master/docs/linux/external_fuzzing_network.md.
[39] Google. 2015. Extract tcp sequence numbers from /dev/net/tun. https://github.com/google/syzkaller/pull/175.
[40] Google. 2015. Syzkaller. https://github.com/google/syzkaller.
[41] Google. 2015. Syzkaller syscall descriptions. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions.md.
[42] Google. 2023. Gemini. https://gemini.google.com/.
[43] Google. 2025. Syzbot. https://syzkaller.appspot.com/.
[44] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 202–215.
[45] HyungSeok Han and Sang Kil Cha. 2017. Imf: Inferred model-based fuzzer. In *CCS 17*. 2345–2358.
[46] Jingmei Hu, Eric Lu, David A Holland, Ming Kawaguchi, Stephen Chong, and Margo Seltzer. 2023. Towards porting operating systems with program synthesis. *TOPLAS* 45, 1 (2023), 1–70.
[47] Jie Hu, Qian Zhang, and Heng Yin. 2023. Augmenting greybox fuzzing with generative ai. *arXiv preprint arXiv:2306.06782* (2023).
[48] Samuel Jero, Endadul Hoque, David Choffnes, Alan Mislove, and Cristina Nita-Rotaru. 2018. Automated Attack Discovery in TCP Congestion Control Using a Model-guided Approach. In *Proc. of NDSS*.

[49] Samuel Jero, Hyojeong Lee, and Cristina Nita-Rotaru. 2015. Leveraging state information for automated attack discovery in transport protocol implementations. In *DSN*. IEEE, 1–12.

[50] Susmit Jha and Sanjit A. Seshia. 2014. Are There Good Mistakes? A Theoretical Analysis of CEGIS. *Electronic Proceedings in Theoretical Computer Science* 157 (July 2014), 84–99. doi:10.4204/eptcs.157.10

[51] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[52] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel.. In *NDSS*.

[53] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *ACM CCS*. 2123–2138.

[54] Junqiang Li, Senyi Li, Gang Sun, Ting Chen, and Hongfang Yu. 2022. SNPSFuzzer: A Fast Greybox Fuzzer for Stateful Network Protocols using Snapshots. *arXiv preprint arXiv:2202.03643* (2022).

[55] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. 2022. μAFL: non-intrusive feedback-driven fuzzing for microcontroller firmware. In *ICSE*. 1–12.

[56] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018).

[57] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In *ESEC/FSE*. 809–814.

[58] Linux Kernel. 2025. KCOV: code coverage for fuzzing. https://docs.kernel.org/dev-tools/kcov.html.

[59] Linux Kernel. 2025. Universal TUN/TAP device driver. https://docs.kernel.org/networking/tuntap.html.

[60] LLVM. 2025. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[61] Dominik Maier, Benedikt Radtke, and Bastian Harren. 2019. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *USENIX WOOT 19*.

[62] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE TSE* (2019).

[63] Alok Mathur, Shreyaan Pradhan, Prasoon Soni, Dhruvil Patel, and Rajeshkannan Regunathan. 2023. Automated test case generation using t5 and GPT-3. In *ICACCS*, Vol. 1. IEEE, 1986–1992.

[64] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *NDSS*, Vol. 2024.

[65] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. 2021. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *USENIX Security 21*. 1683–1700.

[66] Roberto Natella. 2021. StateAFL: Greybox Fuzzing for Stateful Network Servers. *arXiv preprint arXiv:2110.06253* (2021).

[67] Roberto Natella and Van-Thuan Pham. 2021. ProFuzzBench: A Benchmark for Stateful Protocol Fuzzing. In *ISSTA*.

[68] OpenAI. 2022. ChatGPT. https://chatgpt.com/.

[69] Oracle. 2016. kernel-fuzzing: Fuzzers for the Linux kernel. https://github.com/oracle/kernel-fuzzing.

[70] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *USENIX Security 18*. https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[71] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. 2017. Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities. In *USENIX Security*.

[72] Anthony Peterson, Samuel Jero, Endadul Hoque, David Choffnes, and Cristina Nita-Rotaru. 2020. aBBRate: Automating BBR Attack Exploration Using a Model-Based Approach. In *RAID 2020*. USENIX Association, 225–240. https://www.usenix.org/conference/raid2020/presentation/peterson

[73] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *IEEE ICST*.

[74] NCC Group Plc. 2016. TriforceAFL. https://github.com/nccgroup/TriforceAFL.

[75] NCC Group Plc. 2016. TriforceLinuxSyscallFuzzer. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer.

[76] Matthew Renze. 2024. The effect of sampling temperature on problem solving in large language models. In *Findings of the association for computational linguistics: EMNLP 2024*. 7346–7356.

[77] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. 2012. SymDrive: Testing Drivers without Devices. In *OSDI 12*. 279–292.

[78] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. Sok: Prudent evaluation practices for fuzzing. In *IEEE S&P*.

[79] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL:Hardware-Assisted feedback fuzzing for OS kernels. In *USENIX Security 17*. 167–182.

[80] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2022. Nyx-net: network fuzzing with incremental snapshots. In *Proceedings of EuroSys*.

[81] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *USENIX ATC 12*. 309–318.

[82] Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. 2025. The Illusion of Thinking: Understanding the Strengths and Limitations of Reasoning Models via the Lens of Problem Complexity. https://ml-site.cdn-apple.com/papers/the-illusion-of-thinking.pdf

[83] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *NDSS*.

[84] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *CGO*. IEEE, 46–55.

[85] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. 2022. KSG: Augmenting kernel fuzzing with system call specification generation. In *USENIX ATC 22*. 351–366.

[86] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. Healer: Relation learning guided kernel fuzzing. In *SOSP*.

[87] Maolin Sun, Yibiao Yang, Yang Wang, Ming Wen, Haoxiang Jia, and Yuming Zhou. 2023. SMT solver validation empowered by large pre-trained language models. In *ASE*. IEEE, 1288–1300.

[88] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. 2023. SyzDirect: Directed Greybox Fuzzing for Linux Kernel. In *CCS*.

[89] The kernel community. 2025. Introduction to Netlink. https://docs.kernel.org/userspace-api/netlink/intro.html.

[90] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. 2021. SyzVegas: Beating kernel fuzzing odds with reinforcement learning. In *USENIX Security 21*. 2741–2758.

[91] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *ICSE*. 724–735.

[92] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *ICSE*.

[93] Xunguang Wang, Daoyuan Wu, Zhenlan Ji, Zongjie Li, Pingchuan Ma, Shuai Wang, Yingjiu Li, Yang Liu, Ning Liu, and Juergen Rahmel. 2024. Selfdefend: Llms can defend themselves against jailbreaking in a practical manner. *arXiv preprint arXiv:2406.05498* (2024).

[94] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748* (2023).

[95] Jiacheng Xu, He Sun, Shihao Jiang, Qinying Wang, Mingming Zhang, Xiang Li, Kaiwen Shen, Peng Cheng, Jiming Chen, and Charles Zhang. 2025. Demystifying OS Kernel Fuzzing with a Novel Taxonomy. *arXiv:2501.16165* (2025).

[96] Jiacheng Xu, Xuhong Zhang, Shouling Ji, Yuan Tian, Binbin Zhao, Qinying Wang, Peng Cheng, and Jiming Chen. 2024. Mock: optimizing kernel fuzzing mutation with context-aware dependency. In *NDSS*.

[97] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2023. White-box compiler fuzzing empowered by large language models. CoRR abs/2310.15991 (2023).

[98] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2025. Kernelgpt: Enhanced kernel fuzzing via large language models. In *ASPLOS*. 560–573.

[99] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. 283–294.

[100] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In *PLDI*.

[101] Michał Zalewski. 2014. American fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[102] Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2024. How effective are they? exploring large language model based fuzz driver generation. In *ISSTA*. 1223–1235.

[103] Qiang Zhang, Yuheng Shen, Jianzhong Liu, Yiru Xu, Heyuan Shi, Yu Jiang, and Wanli Chang. 2024. ECG: Augmenting Embedded Operating System Fuzzing via LLM-Based Corpus Generation. *TCAD* (2024).

[104] Shenyi Zhang, Yuchen Zhai, Keyan Guo, Hongxin Hu, Shengnan Guo, Zheng Fang, Lingchen Zhao, Chao Shen, Cong Wang, and Qian Wang. 2025. Jbshield: Defending large language models from jailbreak attacks through activated concept analysis and manipulation. *arXiv preprint arXiv:2502.07557* (2025).

[105] Zhiyu Zhang, Longxing Li, Ruigang Liang, and Kai Chen. 2025. Unlocking Low Frequency Syscalls in Kernel Fuzzing with Dependency-Based RAG. In *ISSTA*.

[106] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. 2022. StateFuzz: System Call-BasedState-Aware linux driver fuzzing. In *USENIX Security 22*. 3273–3289.

[107] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. 2021. TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing. In *USENIX ATC 21*. 489–502.

Table A1: Basic Block Coverage Improvement

Fuzzer	Min-Impr	Max-Impr	Average
KnitFuzz	-	-	0
KnitFuzz (no endpoint)	-0.21%	12.95%	5.75%
KnitFuzz (no mutate)	-4.28%	9.79%	1.99%
Syzkaller	10.97%	30.10%	19.13%
MoonShine	13.61%	26.84%	19.38%

Table A2: Branch Coverage Improvement

Fuzzer	Min-Impr	Max-Impr	Average
KnitFuzz	-	-	0
KnitFuzz (no endpoint)	-0.09%	13.33%	5.84%
KnitFuzz (no mutate)	-3.66%	10.43%	2.53%
Syzkaller	7.46%	24.18%	14.12%
MoonShine	8.61%	22.20%	14.59%
Healer	22.85%	38.77%	29.84%

Table B3: Discovered Bugs in `AF_SMC`

Bug	Status	KnitFuzz	Syzkaller	Moonshine	Healer
Deadlock	Reported (by us)	✓	✗	✗	✗
Null-ptr	Reported & Fix	✓	✓	✓	✓

```
static struct sock *smc_tcp_syn_recv_sock(const struct sock *sk /*
    omitted */) {
  struct smc_sock *smc = smc_clcsock_user_data(sk);
  if (READ_ONCE(sk->sk_ack_backlog) + atomic_read(&smc->
      queued_smc_hs) > sk->sk_max_ack_backlog)
    goto drop;
  /* omitted */
}
```

Figure B4: Snippet of `net/smc/af_smc.c`

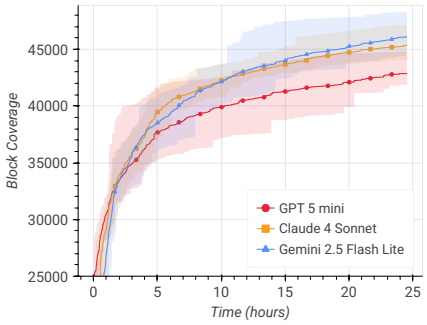


Figure A1: Basic Block Coverage across different LLMs

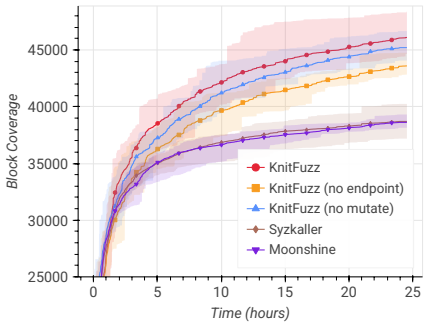


Figure A2: KnitFuzz vs. other fuzzers (BB Cov.)

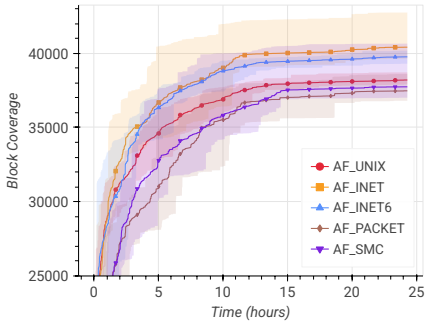


Figure A3: KnitFuzz on different domains (BB Cov.)

Ethics Statement

We carefully evaluated the ethical implications of our research throughout the study. We conducted no experiments on publicly accessible devices or networks, preventing unintended disruptions and data exposure. Potentially harmful vulnerabilities were responsibly disclosed to relevant parties to enable patching before public release. By identifying vulnerabilities early, KnitFuzz improves security and reliability.

Appendix A Additional Evaluation Results

Tables A1 and A2 present the improved coverage of KnitFuzz over all other fuzzers.

We evaluated 3 LLMs for seed generation using code coverage as the metric. Figure A1 plots median coverage (solid lines) with min–max ranges (shaded). Gemini 2.5 Flash Lite achieves the highest coverage, outperforming Claude 4 Sonnet and GPT 5 mini.

Figure A2 shows the achieved basic block coverage by KnitFuzz compared to other fuzzers. Healer is excluded from this comparison, as it reports only branch coverage. Figure A3 shows the achieved basic block coverage by KnitFuzz across 5 domains.

Appendix B Summary of Bug Discovery

Table B3 summarizes the discovered bugs and their current status. Figure B4 shows the null pointer dereference discovered by KnitFuzz in `net/smc/af_smc.c`.

Appendix C PoC of the Reported Bug

Minimal reproducible seed of the reported bug is available at https://github.com/syne-lab/knitfuzz/blob/main/results/good/1296.c.

Appendix D Porting of Related Work

- Syzkaller[40]: We only enabled a subset of network related system calls in order to match the system calls we are actually testing. Moonshine and Actor's Syzkaller components have the same modification as well. An additional parameter `stat.Console` to `pkg/corpus/corpus.go` in Syzkaller and moonshine to output the signal (edge coverage). This should not affect Syzkaller's performance as the manager and executors are decoupled.

Table E4: Distribution of Open Bugs

Reproducer Type	Count	Percentage
C reproducer	787	53.1%
Syz Reproducer	124	8.4%
No Reproducer	572	38.6%
Total	1483	100.0%

Table E5: Distribution of Invalid Bugs

Reproducer Type	Count	Percentage
C Reproducer	1766	10.5%
Syz Reproducer	444	2.7%
No Reproducer	14540	86.8%
Obsoleted	13566	81.0%
Total	16750	100%

- Moonshine [70]: Migrate to Go Modules so that it can be compiled with the latest golang compiler. The directory structure is changed as it was using the deprecated vendoring tools. Import paths are changed accordingly. `go.mod` and `go.sum` are locked against the commit hash mentioned in moonshine's README.
- Healer [86]: The `syz_wrapper` component cannot be compiled any more due to the more restrictive check of the latest c compiler. We port the changes from the latest Syzkaller to fix the part `common_usb*.h` that produce warnings. Other parts within `syz_wrapper` remain unchanged. Additionally, to enable a fair comparison, we limit the total system calls to those were used in KnitFuzz, these restriction is a one time cost (less than 1 second) that are introduced before healer entering the real fuzzing stage. We also fixed the QEMU version check in healer.
- Actor [36]: We port actor's kernel patches to the latest one. We fixed the uio ivshmem kernel module compilation error by using the corresponding helper function. Furthermore, we fix the invalid argument error on mmapping uio ivshmem by changing the target PAT memory attributes from cached to noncached. Kernel build dirs are changed accordingly as the kernel is built within the docker container. However, it still aborts when envtrack got illegal address and fed into `addr2line`. Therefore, we exclude it from fuzzer comparison.
- kAFL [79]: kAFL requires Intel PT to work. We are running on the AMD platform.
- FuzzNG [16]: FuzzNG recommends using Intel VT-x CPUs to run the fuzzer and uses Intel specific argument to run the QEMU instance. We are running on the AMD platform. Despite that, we tried to port FuzzNG's kernel patches to the latest one. However, it fails due to a large refactoring in lib/usercopy.c. Furthermore, FuzzNG does not have a configuration for network stack and by default exclude linux/net/ folder from coverage collection. Therefore, we exclude it from fuzzer comparison.
- SyzDirect [88]: SyzDirect requires a set of pre-selected bugs to work. The artifact does not have them available out-of-box. The fuzzer needs xls format to run, while the dataset is provided in pdf and missing critical fields for preparing and fuzzing. Therefore, we exclude it from fuzzer comparison.
- Fuzzers [18, 19, 85, 98, 105] that generate Syzkaller specifications: They are excluded from comparison because Syzkaller specifications generated were contributed to the Syzkaller upstream.
- Mock [96]: Mock requires neural network model to work. However, neither model weights nor training corpus is available. Therefore, we excluded it from fuzzer comparison.

Appendix E Syzbot Statistics

Table E4 and E5 shows statistics collected from syzbot dashboard [43].