SeedSmith: LLM-Driven Seed Synthesis for Directed Fuzzing

Junmin Zhu*
UC Santa Barbara
California, USA
junmin@ucsb.edu

Siyu Liu*
Arizona State University
Arizona, USA
sliu274@asu.edu

Jie Hu
Arizona State University
Arizona, USA
jiehu12@asu.edu

Fabio Gritti
UC Santa Barbara
California, USA
degrigis@cs.ucsb.edu

Ati Priya Bajaj
Arizona State University
Arizona, USA
atipriya@asu.edu

Hulin Wang
Arizona State University
Arizona, USA
hwang551@asu.edu

Wenbo Guo
UC Santa Barbara
California, USA
henrygwb@ucsb.edu

Tiffany Bao
Arizona State University
Arizona, USA
tbao@asu.edu

Christopher Kruegel
UC Santa Barbara
California, USA
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara
California, USA
vigna@ucsb.edu

Abstract

Directed fuzzing steers fuzzers toward user-defined sink functions to identify vulnerabilities, but it frequently fails to trigger crashes even after long campaigns. We identify two challenges that prevent directed fuzzers from exposing crashes: incomplete static analysis of indirect calls, which leaves reachable paths invisible to distance-based guidance, and lack of semantic guidance for crash preconditions, which blind mutation cannot satisfy within practical time budgets. A natural intervention point is the initial seed corpus: seeds that encode the right control-flow path and satisfy key crash preconditions shift fuzzing from blind exploration to local refinement. Existing seed generation approaches address neither: grammar-based and format-driven methods produce structurally valid inputs with no sink awareness, while LLM-based methods either lack sink targeting or inherit static analysis limitations through one-shot prompting. We present SeedSmith, an agentic LLM pipeline that replicates a security analyst's workflow: starting from a sink, SeedSmith iteratively explores the codebase, resolves indirect calls, identifies crash preconditions, and synthesizes concrete inputs that satisfy them. Because SeedSmith operates as a seed generation front-end, its seeds are fuzzer-agnostic and improve any downstream mutation-based fuzzer without modification. On Magma, fuzzers using SeedSmith seeds achieve geometric mean crash-time speedups of 11.51× (AFL++) to 14.66× (AFLGo) over default seeds. On ARVO, SeedSmith enables fuzzers to trigger 16 previously unreachable bugs spanning 10 projects with diverse input formats.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

CCS Concepts

• **Security and privacy** → **Software and application security**;
• **Computing methodologies** → *Artificial intelligence.*

Keywords

directed fuzzing, seed generation, large language models, vulnerability discovery

ACM Reference Format:
Junmin Zhu*, Siyu Liu*, Jie Hu, Fabio Gritti, Ati Priya Bajaj, Hulin Wang, Wenbo Guo, Tiffany Bao, Christopher Kruegel, and Giovanni Vigna. 2018. SeedSmith: LLM-Driven Seed Synthesis for Directed Fuzzing. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 17 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Directed fuzzing [8, 10, 17, 24, 46, 51, 59] is a powerful technique for uncovering vulnerabilities in specific code regions, with proven applications in static-analysis report verification [8], crash reproduction [59], and patch testing [51]. Unlike coverage-guided fuzzing, which broadly explores a program's state space, directed fuzzers focus their efforts on user-defined *sink functions*, which are locations likely to harbor security-critical flaws. Yet despite this focus, directed fuzzers frequently fail to trigger vulnerabilities even after long campaigns, even though prior work has significantly advanced their scheduling [10] and mutation [46] strategies.

Two challenges explain why directed fuzzers frequently fail to trigger crashes. First (**C1**), many directed fuzzers [10, 24, 44] rely on call graph information from static analysis to compute the distance metrics used to drive the exploration process. However, indirect

calls, pervasive in real-world C/C++ programs through function pointers and virtual dispatch, are notoriously difficult to resolve statically [45]. As a result, when call edges are missing, the fuzzer's distance computation is incomplete, leaving entire reachable paths invisible to the guidance algorithm [32]. Second (**C2**), even when the fuzzer reaches the sink, triggering the crash often requires satisfying precise input-level conditions, such as specific field values, format constraints, or data-flow dependencies. These conditions are invisible to coverage-based feedback, and blind byte-level mutation is unlikely to satisfy all constraints simultaneously within practical time budgets [14].

The natural intervention point for both challenges is the *initial seed corpus*: a seed that already encodes the correct control-flow path and satisfies key crash preconditions reduces the fuzzer's task from blind exploration to local refinement. Yet existing seed generation techniques fall short. Grammar-based [9, 49], format-specification-driven [18], and general LLM-prompting approaches such as Fuzz4All [50] produce structurally valid inputs, satisfying the format-validity component of **C2**, but are sink-agnostic: they do not reason about a specific target, so they they do not address **C1** and leave sink-specific preconditions unaddressed, such as which particular field values or data-flow patterns trigger the crash. ISC4DGF [52] and Magneto [61] do target specific sinks, but rely on one-shot prompting with static analysis summaries, directly inheriting their incompleteness and providing no mechanism to iteratively reason about crash preconditions. What is needed is a seed generation approach that actively explores the codebase, resolves indirect calls that static analysis misses, and reasons about the input conditions required to manifest the bug.

In fact, when trying to reach a vulnerable code location, a skilled security analyst does not rely on a precomputed call graph or a single-shot prompt: starting from a sink function, they iteratively explore the codebase, trace call paths, resolve indirect calls by reading function bodies and type definitions, identify the conditions required to trigger the crash, and synthesize a concrete input that satisfies them. Recent advances in LLMs allow us to automate this workflow: the demonstrated ability of LLMs to comprehend code structure, reason about data flows, and understand format specifications [40, 58] maps directly onto each step of the analyst's process, without the incompleteness that limits static analysis.

Given this insight, we present SeedSmith, a system that addresses both challenges by using an LLM agent to replicate the workflow of a security analyst performing manual seed construction.

SeedSmith is designed as a two-stage agentic pipeline. An *Analysis Agent* reconstructs the execution path from harness to sink under a strict context budget, supported by two design choices that keep the search tractable. First, a *path optimization* step compresses the over-approximated call graph into a single linearized path: prefix and suffix segments shared across alternative routes are preserved verbatim, while their divergent middles are collapsed into a single connector node that the agent can expand on demand. The agent thus starts from a compact outline of the reachability structure rather than the full set of paths. Second, its code-search tool is *context-aware*: rather than returning a fixed-line window around a search result, the tool inspects the match site and returns the enclosing function body, type definition, or configuration block, as appropriate, so each query yields a semantically complete unit rather than a truncated fragment. Iterative queries against this tool enable the agent to resolve indirect calls (**C1**) and extract crash preconditions (**C2**) that static analysis misses, and to emit a structured report on convergence.

The *Seed Generation Agent* then consumes this report and produces Python scripts that programmatically construct test inputs, using format libraries where raw bytes cannot satisfy structural constraints. Generated seeds are validated against a sanitizer-instrumented binary in a refinement loop, with execution feedback used to correct them. Separating analysis from generation also serves a deliberate cost/capability trade-off: the heavyweight code-reasoning model is invoked only during the exploration (analysis) stage, with a cheaper model handling seed generation.

We evaluate SeedSmith on 23 Magma bugs and 115 ARVO challenges across 26 projects, to our knowledge, the largest evaluation of LLM-based seed generation for fuzzing to date. On Magma, fuzzers using SeedSmith seeds achieve geometric mean crash-time speedups of 11.51× (AFL++) to 14.66× (AFLGo) over default seeds. Notably, AFL++ with SeedSmith outperforms AFLGo at exposing targeted crashes, showing that SeedSmith can give a general-purpose fuzzer directed-fuzzing capability. On ARVO, fuzzers using SeedSmith seeds trigger 16 bugs that AFLRun and AFL++ with default seeds never trigger, spanning 10 projects with diverse input formats.

In summary, we make the following contributions:

- We identify two fundamental challenges, incomplete static analysis of indirect calls (**C1**) and lack of semantic guidance for crash preconditions (**C2**), that limit both directed fuzzers and existing seed generation techniques for sink-oriented vulnerability discovery.
- We design and implement SeedSmith, an agentic LLM pipeline that iteratively explores target codebases to generate structurally valid, sink-targeted seeds, addressing **C1** and **C2** without modifying the downstream fuzzer.
- We conduct a large-scale evaluation on 23 Magma bugs and 115 ARVO challenges across 26 projects, demonstrating significant crash-time speedups and new crash discovery across four fuzzers.

2 Background

Fuzzing. Fuzzing is a dynamic software testing technique that automatically generates inputs to discover bugs in target programs. At the start of a fuzzing campaign, the fuzzer is seeded with at least one initial test case. It then repeatedly selects a seed from the corpus, applies mutations to produce new inputs, and executes the target program. Inputs that trigger new behavior, such as covering previously unseen code, are retained in the corpus for further mutation. Directed fuzzing [10, 12, 17] extends this workflow by steering execution toward a predefined set of target locations, known as sinks. Rather than maximizing overall coverage, directed fuzzers assign higher priority to seeds whose execution traces are estimated to be closer to the sink, using distance metrics computed over a control-flow graph constructed via static analysis. This makes directed fuzzing especially effective when prior knowledge has identified specific code locations as likely vulnerability sites, such as recently

patched functions or sinks flagged by static-analysis tools. However, the effectiveness of distance-based guidance depends critically on the completeness of the underlying static analysis: when call edges are missing, distance metrics are unreliable, and the fuzzer cannot steer execution toward paths it cannot see. Recent work uses LLMs to augment the fuzzer's feedback signal in this setting: Locus [62] employs an LLM agent to synthesize predicates that are instrumented into the target program, steering mutation toward predicate-satisfying inputs without modifying the seed corpus.

Seed Generation. The quality of the initial seed corpus significantly impacts fuzzing effectiveness, especially for programs that accept highly structured inputs such as images, documents, or protocol messages. Traditional approaches generate seeds using learned grammars [9], data-driven probabilistic models [49], or binary format specifications [18], but require explicit grammar knowledge and do not incorporate information about the target vulnerability. More recently, LLMs have been applied to seed generation. Fuzz4All [50] prompts LLMs with language specifications to generate diverse inputs, but is not sink-targeted. ISC4DGF [52] prompts an LLM in a single shot with a static-analysis summary of the C/C++ target to generate seeds tailored to a directed-fuzzing sink. Magneto [61] targets Java dependent-library exploitation: it decomposes a known call chain edge by edge and invokes the LLM once per edge on a statically-sliced focal-code window to produce a JSON seed template. Both ISC4DGF and Magneto pre-compute static analysis and push the resulting summary or slice into the LLM's prompt, inheriting whatever imprecisions (such as missing indirect calls and data-flow facts) that analysis produces. SeedSmith differs by employing an *agentic* LLM pipeline that iteratively queries the codebase through a code-search tool and refines its understanding over multiple steps, letting the LLM resolve indirect calls and infer crash preconditions on demand rather than from a fixed static-analysis snapshot.

LLM Agents for Code Reasoning. Large language models are pre-trained on massive corpora that include billions of lines of code, giving them strong capabilities for code comprehension, data-flow reasoning, and format specification understanding without task-specific fine-tuning [13, 35]. When equipped with tools such as code search or file retrieval, LLMs can operate as agents that iteratively query an external environment, observe results, and refine their reasoning over multiple steps [53]. This agentic paradigm is well-suited to tasks that require building up context incrementally, such as tracing an execution path through a large codebase, resolving indirect calls by reading type definitions and function bodies, and identifying the precise conditions under which a vulnerability manifests. SeedSmith exploits these capabilities to replicate the workflow of a security analyst constructing a crash-triggering input from scratch.

3 Motivation

We motivate SeedSmith with two real-world bugs that expose the limitations of state-of-the-art directed fuzzers, each isolating one of the challenges from Section 1: An nginx bug for **C1** and an openjpeg bug for **C2**.

3.1 C1: Indirect Calls Break Static Guidance

Static-analysis-based distance metrics assume that the call graph is sufficiently complete to measure proximity to the sink. In practice, indirect calls through function pointers and virtual dispatch routinely violate this assumption, leaving entire reachable paths invisible to the fuzzer's guidance algorithm.

```
1 @@ -70,7 +70,7 @@ ngx_sendfile_r(ngx_connection_t *c, ngx_buf_t *
      file, size_t size)
2     lseek(file->file->fd, 0, SEEK_SET);
3 +   rev = ngx_alloc(NGX_SENDFILE_R_MAXSIZE, c->log);
4 -   rev = ngx_alloc(size, c->log);
5     if ( rev == NULL ) {
6         return NGX_ERROR;
7
```

(a) Git diff of the injected bug

```
GET / HTTP/1.1
Host: localhost
Range: bytes=-r, 0-614
```

(b) PoV: the `-r` flag in the `Range` header triggers the overflow.

Figure 1: Motivating example for C1: an `nginx` bug invisible to static call-graph guidance.

Consider the bug shown in Figure 1a, injected into nginx [41] during the DARPA AIxCC competition [1]. The vulnerability is a heap-buffer overflow in `ngx_sendfile_r`, which allocates a fixed-size buffer using the hard-coded constant `NGX_SENDFILE_R_MAXSIZE` (Line 3) instead of the actual resource size. Triggering it requires the `-r` flag to appear at the correct position in the HTTP Range header (Figure 1b), which routes execution into `ngx_sendfile_r` through a chain of function pointers.

AFLRun [44] fails to trigger this bug or even reach `ngx_sendfile_r` after 24 hours. The reason is that `ngx_sendfile_r` is reachable only through indirect calls that static analysis cannot resolve [45]: no call edge connects the harness to the sink in the computed call graph, so the distance metric assigns the sink infinite distance and the fuzzer receives no gradient toward it. No amount of mutation can compensate for a missing call edge; the fuzzer simply cannot see the path.

3.2 C2: Crash Preconditions Defeat Blind Mutation

Even when the fuzzer successfully reaches a sink, triggering the crash may require satisfying a conjunction of precise input-level conditions that coverage-based feedback cannot distinguish from ordinary reachability.

Consider the bug in openjpeg shown in Figure 2a. The crash site is inside `opj_j2k_decode_tile` (Line 9), but it is guarded by six tile-geometry predicates (Lines 1–6) that must all hold simultaneously: the tile grid must begin at the origin, the output window must be aligned to the codec's tile size, and the dimensions must match exactly. Only when all six conditions are satisfied does execution reach the vulnerable call.

```
if (p_j2k->m_cp.tw == 1 && p_j2k->m_cp.th == 1 &&
    p_j2k->m_cp.tx0 == 0 && p_j2k->m_cp.ty0 == 0 &&
    p_j2k->m_output_image->x0 == 0 &&
    p_j2k->m_output_image->y0 == 0 &&
    p_j2k->m_output_image->x1 == p_j2k->m_cp.tdx &&
    p_j2k->m_output_image->y1 == p_j2k->m_cp.tdy) {
    /* read tile header; return OPJ_FALSE on failure */
    if (!l_go_on ||
            ! opj_j2k_decode_tile(p_j2k, l_current_tile_no, NULL,
      0, p_stream, p_manager)) {
        opj_event_msg(p_manager, EVT_ERROR, "Failed to decode tile
       1/1\n");
        return OPJ_FALSE;
    }

```

(a) Crash site, gated by six tile-geometry predicates.

```
from PIL import Image, ImageDraw
# Create a 200x200 image
img = Image.new('RGB', (200, 200), color='white')
draw = ImageDraw.Draw(img)
# Draw a blue circle
draw.ellipse([50, 50, 150, 150], fill='blue')
# Draw a red rectangle
draw.rectangle([75, 75, 125, 125], fill='red')
# Save as JPG
img.save('poc.txt')

```

(b) Python script example emitted by SeedSmith.

Figure 2: Motivating example for C2: an `openjpeg` crash whose six tile-geometry predicates must all hold.

AFLRun correctly prioritizes inputs that reach the enclosing function, but it never triggers the crash. The fuzzer's distance computation provides no signal for satisfying the structural constraints: A seed that reaches the function through the correct path looks identical to one that satisfies all six predicates, from the coverage-feedback perspective. Blind byte-level mutation is unlikely to satisfy all constraints simultaneously within practical time budgets [14], and the vulnerable basic block remains unexecuted throughout the entire 24-hour campaign.

3.3 Implications for Seed Generation

These two cases reveal a shared limitation that cannot be overcome by improving the fuzzer's scheduler or mutation strategy alone. When the call graph is incomplete, no distance metric can guide execution toward an unknown path (C1). When the crash requires satisfying complex structural predicates, byte-level mutation cannot discover the required input within practical time budgets (C2). The natural intervention point is the initial seed corpus: A seed that already encodes the correct control-flow path and satisfies key crash preconditions reduces the fuzzer's task from blind exploration to local refinement, making both challenges tractable.

4 Design and Overview

SeedSmith operates by generating a set of high-quality initial seeds aimed at rapidly triggering potential crashes in targeted code regions. This is achieved by harnessing LLMs' ability to comprehend code structure and data flow, enabling SeedSmith to produce semantically meaningful inputs that sidestep the indirect call resolution limitations inherent in static-analysis-dependent directed fuzzers. More importantly, because SeedSmith augments the initial corpus with these seeds rather than replacing any fuzzer component, it is fuzzer-agnostic: the generated seeds can be readily applied to improve both directed fuzzers and coverage-guided fuzzers. Figure 3 presents an overview of SeedSmith's pipeline, which consists of three components described below.

❶ **Target Preparation**. Taking the target program and sink function as input, this component performs lightweight static analysis and instrumentation to extract program structure and build a queryable knowledge base for the downstream LLM agents.

❷ **Agentic Code Exploration**. Taking the harness code, sink function, and knowledge base as input, this component employs an LLM agent to explore the codebase and produce a comprehensive analysis report capturing the control-flow paths, data-flow dependencies, and input format requirements needed to reach the target.

❸ **Agentic Seed Generation**. Taking the analysis report as input, this component employs a lightweight LLM agent to generate a set of concrete, semantically meaningful test inputs as the final seed corpus.

The resulting corpus is then handed off to a downstream fuzzer. Because SeedSmith operates as a seed generation front-end, it is compatible with any mutation-based fuzzer engine, enabling more directed exploration and faster crash discovery without modifying the fuzzer itself.

4.1 Target Preparation

SeedSmith assumes the target is a C/C++ project with available harnesses (e.g., as provided by OSS-Fuzz), and performs three preparation steps:

1) Project Indexing. SeedSmith identifies all functions and their boundaries within the project, building an index that maps function names to their locations and source code for efficient retrieval during downstream code exploration.

2) Call Path Identification. SeedSmith extracts a static call graph from CodeQL. Direct calls are resolved soundly; indirect calls are resolved by type-signature matching, linking each function-pointer call to every signature-compatible function whose address is taken somewhere in the codebase. This best-effort resolution may both over-approximate, when multiple functions share a parameter signature, and miss edges, when types are obscured by casts or generic pointers. SeedSmith deliberately does not require the call graph to be sound: the analysis agent later recovers missing edges through tool-assisted exploration (Section 4.2). From this graph, SeedSmith enumerates candidate paths between the harness and the sink. Since providing all paths to the LLM would exceed its context limit, SeedSmith applies a *path optimization* step to produce a compact yet informative representation. Specifically, the algorithm identifies common prefix and suffix segments shared across paths, and merges the divergent middle portions into a single connector node that indicates the existence of multiple branches (illustrated in the *Call Graph* panel of Figure 3). The result is a linearized path that preserves the essential control flow structure while remaining within

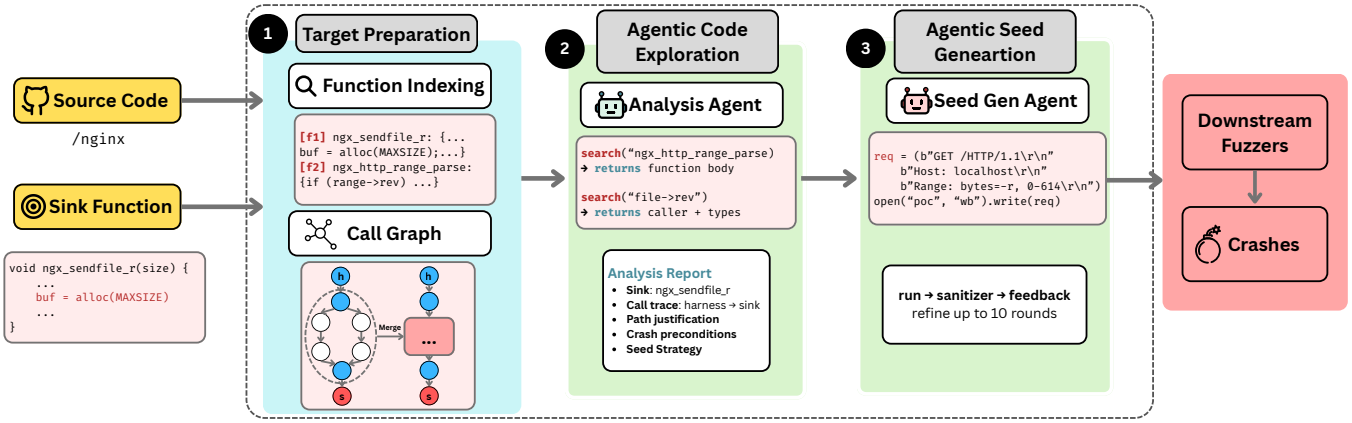

Figure 3: SeedSmith overview. The system processes a C/C++ project through three main stages: ❶ Target preparation indexes functions, builds call graphs, and instruments the code with sanitizers. ❷ The Analysis Agent explores the codebase using tool-assisted retrieval to understand how to reach and crash the sink function. ❸ The Seed Generation Agent creates concrete test inputs based on the analysis, which serve as the initial corpus for any downstream fuzzer.

the token budget, with the agent retaining the option to explore specific branches on demand via tool-assisted retrieval.

3) Project Building. SeedSmith compiles the project with a sanitizer (ASAN, MSAN, or UBSAN) to enable crash detection during seed validation; the sanitizer is taken from the project's existing build configuration.

4.2 Agentic Code Exploration

To address the limitations of conventional directed fuzzers, namely incompleteness of static analysis (**C1**) and semantic information loss (**C2**), SeedSmith employs an LLM-based *Agentic Code Exploration* component. Rather than relying on static analysis alone, an LLM agent iteratively reasons over the identified call paths to understand how the the vulnerability at the sink can be reached and triggered from the harness, ultimately producing a comprehensive analysis report that guides the subsequent agentic seed generation.

Initially, the analysis agent receives three inputs to begin its analysis.

(I1) Harness code. The fuzzing entry point of the target project (e.g., from OSS-Fuzz), serving as the starting point for reasoning about execution paths to the sink.

(I2) Sink function code. The complete source code of the target function of interest, which may be flagged by a developer, a security patch, or a static analysis tool.

(I3) Static call path. The linearized harness-to-sink path produced in step ❶. Because the underlying call graph is not sound, SeedSmith treats this as a starting hint rather than ground truth: the path may be (1) empty, if no path exists due to indirect calls or unreachability; (2) a single direct path, if exactly one path is found; or (3) a linearized representation that collapses multiple paths via shared prefix/suffix segments and connector nodes for divergent middles (Section 4.1).

Tool-Assisted Context Retrieval. The initial inputs (*I1-I3*) rarely suffice to generate valid crash seeds. Even when a call path exists, it does not capture the data-flow dependencies and conditional checks that determine whether the sink can be reached and crashed. Human security researchers address this by iteratively examining the codebase and searching for function definitions, tracing variable assignments, and understanding control-flow conditions. We replicate this process by equipping the agent with a code searching tool.

The search tool accepts keywords or patterns from the LLM and returns contextually enriched results based on the type of match. Rather than returning raw matches with fixed surrounding lines, the tool applies context-aware retrieval rules:

- *Function match*: If a match occurs inside a function body, the tool returns the complete function source code, giving the agent full visibility into how the matched symbol is used.
- *Global variable match*: The tool returns all global declarations in the file, exposing related state that may affect the execution path.
- *Type or structure match*: For matches in source files that correspond to structures, type definitions, or associated comments, the tool returns the relevant declarations with surrounding context.
- *Non-source file match*: For configuration files (`.conf`, `.ini`, `.yaml`), the tool returns the entire file; for other non-source files, it returns a localized window around the match.

This context-aware retrieval ensures the agent receives semantically meaningful units of code rather than arbitrary line ranges. For example, when searching for `file->rev`, if the match appears within `ngx_linux_sendfile()`, the tool returns the entire function's source code to enable analysis of how the variable is used. When searching for a constant like `NGX_SENDFILE_MAXSIZE`, the tool returns surrounding `#define` statements that provide context about buffer sizes and related configuration.

Through iterative queries, the agent incrementally accumulates the data- and control-flow dependencies along the execution path. Each search result may reference new functions, variables, or control-flow conditions, which become targets for subsequent searches. This process continues until the agent identifies a complete chain from harness to sink with all necessary preconditions.

For instance, in **C1-Example**, the agent identifies the omitted function `ngx_http_range_parse()`, traces how `range->rev` is assigned to `file->rev`, and discovers the critical control-flow condition `if (file->rev)` that determines whether execution reaches the vulnerable code path. The agent also interprets comments and structures to understand input format requirements, discovering that the `-r` flag in the HTTP Range header can trigger the vulnerability. We will show more details about how the analysis agent explores the codebase and finds crashes in Section 6.3.

Analysis Report Generation. Upon completing its exploration, the agent consolidates its findings into a structured analysis report comprising six components: (1) the sink function's location and complete source code; (2) the available harness entry points; (3) one complete path (call sequence) from the harness to the sink, annotated with the relevant code snippet at each step; (4) a justification for the selected path over alternatives; (5) a detailed analysis of the conditions required to trigger the crash; and (6) a concrete seed generation strategy, including an example Python script. This report serves as the sole input to the subsequent Seed Generation Agent (❸), providing all information necessary to produce crash-triggering inputs without further code exploration.

4.3 Agentic Seed Generation

The seed generation agent leverages the *Analysis Report* to produce testing inputs (seeds) that attempt to trigger the vulnerability at the sink (*I2*) via the provided harness (*I1*).

Instead of generating raw binary inputs directly, the seed generation agent emits Python scripts that construct the seeds programmatically. This design addresses the file structure challenge. Complex file formats such as images, PDFs, or compressed archives require specialized libraries and must satisfy strict structural constraints like headers, checksums, and metadata that LLMs cannot reliably produce as raw bytes. The agent can leverage a set of Python packages, e.g., `PIL`, to create the target file structure.

Once generated, SeedSmith executes the Python script in an isolated environment to produce raw input seeds. To maximize the likelihood of triggering a crash, seed generation follows an iterative refinement loop of up to ten rounds. In each round, the generated script is executed, and the resulting seeds are tested against the sanitizer-instrumented target program. The loop terminates early if the target crashes on a generated seed; otherwise, the execution feedback (sanitizer output, exit code, and stderr) is passed back to the seed generation agent, which uses it to revise the script for the next round. If no crash is detected after ten rounds, all seeds produced across rounds are collected as the final corpus. Figure 2b shows an example of the script produced.

The resulting seeds are fuzzer-agnostic and can be directly used as an initial corpus for any downstream fuzzer. Because the seeds already encode complex reachability conditions such as specific control-flow branches, data-flow dependencies, and input format requirements, they significantly reduce the exploration burden on the fuzzer. This benefit holds regardless of the fuzzer's underlying strategy, whether it is directed guidance (e.g., AFLGo [17]), coverage-guided mutation (e.g., AFL++), or otherwise.

5 Implementation

SeedSmith was implemented in Python for the core functionality, and it integrates both off-the-shelf and custom tools throughout the various stages of the pipeline.

Target Preparation. To prepare the target, we leverage off-the-shelf analysis tools. In particular, we use `CodeQL` [3] to build the initial call graph (Section 4.1), and we store such information in a `Neo4j` [2] graph database to be queried during runtime from the agent at step ❶. For *Project Indexing*, we implemented a custom C indexer for the extraction of functions and globals. This indexer is similar to `tree-sitter` [5], but extends it with better function boundaries and macro-resolution capabilities.

Agentic Code Exploration. We use the `LangChain` framework [4] to implement the agents in steps ❷ and ❸. Our LLM backends are provided by Anthropic and OpenAI. We use `Claude Sonnet 4` for the *Agentic Code Exploration* with default inference parameters (temperature, top-p) because of its state-of-the-art code understanding capability and relatively cheap pricing at the time of evaluation. We implement the context-aware search tool described in Section 4.2 using the *grep* tool directly. While the interface exposed to the LLM is intentionally simple (keyword-based search), the tool internally applies the retrieval rules to enrich raw matches before returning them to the agent.

Specifically, function boundaries and global variable scopes are resolved using the project index built during Target Preparation, and type definitions are extracted from header files with `X=10` lines of surrounding context for structural matches (*type or structure match*), and `Y=2` lines for non-source file matches. These values were chosen empirically: `X=10` suffices to capture most `struct`/`typedef` definitions together with adjacent field comments and related `#defines` in typical C/C++ headers, while `Y=2` provides enough context for key-value entries in configuration files without flooding the agent's context window.

To manage context efficiently, the search tool implements several optimizations. First, it caches matched lines and enforces deduplication: each unique code snippet appears at most once in the tool output, even if matched multiple times. Second, if the combined content from all results for a single invocation exceeds 50K tokens, the tool discards the results and returns an informative error message, prompting the agent to issue a more targeted query. Returning a truncated subset would risk biasing the agent toward whichever results appear first, whereas an empty result with an explanation preserves the agent's ability to reason about the failure and refine its search. Third, the agent implements an early termination heuristic: if no matches are found for 30 consecutive tool calls, the analysis halts, and the agent generates its final report. Since the optimal configuration depends on the specific target under analysis, the values above are the defaults that worked well in our experiments.

Agentic Seed Generation. We use `gpt-4.1-mini` for the *Agentic Seed Generation*, because of its low-cost but still powerful code generation ability. We set the Seed Generation Agent to run ten times. The generated seeds are validated by executing them against the instrumented binary built during Target Preparation; execution feedback (e.g., unintended crashes or failure to reach the sink) is fed back to the agent for iterative refinement.

6 Evaluation

We conduct a comprehensive evaluation of SeedSmith to demonstrate its effectiveness as an LLM-driven seed generator for directed fuzzing. Our evaluation answers the following research questions:

RQ1: How effective is SeedSmith at generating seeds that crash target sink functions compared to baseline approaches?
RQ2: How does the LLM's reasoning ability help SeedSmith discover crashes faster than state-of-the-art fuzzers?
RQ3: What is the contribution of each component in the SeedSmith design?

We answer **RQ1** by comparing crash counts and time-to-crash for downstream fuzzers using SeedSmith seeds versus default seeds on ARVO and Magma, and additionally against Locus [62] on Magma (Section 6.2). For **RQ2**, we manually investigate how LLM reasoning enables SeedSmith to expose crashes significantly faster than baseline fuzzers (Section 6.3). For **RQ3**, we present an ablation study quantifying the contributions of the LLM seed generator, the scan strategy, and the control-flow support (Section 6.4).

6.1 Experiment Design

Dataset. We use two benchmarks for a comprehensive evaluation of SeedSmith: Magma and ARVO.

Magma [23] is a widely used fuzzing benchmark. For our evaluation, we select 23 bugs spanning 8 projects, representing the exact intersection of the datasets used in AFLRun and LibAFLGo [20]. This intersection was chosen deliberately: it combines the high vulnerability complexity characteristic of AFLRun's benchmark with the well-defined, confirmed-reachable target locations rigorously validated by LibAFLGo. Each bug is instrumented with a unique canary check; a crash is classified as intended when the fuzzer's input triggers that canary. We run each configuration for 24 h × 10 independent trials.

To evaluate real-world applicability, we additionally use *ARVO* [37], a dataset of over 6,000 reproducible real-world memory vulnerabilities across more than 250 open-source projects, discovered by OSS-Fuzz. For each bug, ARVO provides a Docker image with a proof-of-concept (PoC) crashing input. We initially drew a random sample of 200 targets from ARVO to keep the evaluation computationally tractable. To ensure a fair comparison and proper execution of directed fuzzing, we filtered out targets that failed to compile under AFLRun or AFL++, yielding a final set of 115 bugs across 26 projects. A crash is classified as intended when the deduplication token of a fuzzer-generated input matches that of the PoC. We run each configuration for 24 h × 10 trials.

Downstream Fuzzers. To demonstrate that SeedSmith seeds are fuzzer-agnostic, we evaluate them with multiple downstream fuzzers. On Magma we use AFL++(v4.30c), AFLGo [10], AFLRun [44], and FairFuzz [28], covering both directed and coverage-guided strategies. Although AFL++ is designed as a general-purpose fuzzer, prior work has shown that it can outperform existing directed fuzzers at exposing crashes at a target location [44, 56], a pattern we also observe in our evaluation (see Section 6.2.1). AFLGo is a well-known directed fuzzer commonly used as a baseline, FairFuzz optimizes for rare-branch coverage, and AFLRun is one of the latest directed fuzzers that significantly outperformed eight competitors in its own evaluation. On ARVO, we use AFL++ and AFLRun, selected for their compatibility with OSS-Fuzz projects.

Metrics. For each (fuzzer, seed-configuration) pair we measure the *time to trigger* the intended crash, computed as the Restricted Mean Survival Time (RMST) [23] over the 10 trials. Statistical significance is assessed with one-sided Mann–Whitney U tests; we report p-values throughout. All speedup figures reported in this paper are *geometric means* of per-bug RMST ratios (baseline / treatment). When only the baseline or only the treatment times out, we substitute 24 h for its RMST so the ratio remains finite; bugs where both time out are excluded since the ratio is undefined. In all trigger-time tables, **bold** marks the fastest configuration per bug per fuzzer (or the faster of the two in the ablation tables), **T.O.** denotes no crash within 24 h, and **1-shot** denotes a crash from the LLM-generated seed before any fuzzer mutation (counted as 1 s when computing speedups).

Environment. We conducted experiments on Ubuntu 20.04.6 LTS machines, dedicating one core of an Intel Xeon E5-2670 v2 (2.50 GHz) to each fuzzing campaign.

Seed Configurations. For each fuzzer we evaluate four seed configurations: *No-seed* (empty initial corpus), *Default* (OSS-Fuzz seeds only), *SS-only* (SeedSmith-generated seeds only), and *SS-combined* (Default + SS-only). This design isolates the effect of SeedSmith seeds: improvements under SS-only or SS-combined over Default and No-seed are directly attributable to the quality of SeedSmith-generated seeds. Unless otherwise noted, "SS" and "SeedSmith seeds" in the prose refer to *SS-combined*; we use the other configurations only when discussing them explicitly.

6.2 RQ1: Crash Efficiency

We measure crash counts and time-to-crash with SeedSmith seeds versus Default seeds across all four fuzzers on Magma and across AFL++ and AFLRun on ARVO. Among prior LLM-augmented directed-fuzzing systems, ISC4DGF [52] is not open-sourced and Magneto [61] targets Java dependent libraries, neither of which can be run on our C/C++ benchmarks. We therefore additionally compare against Locus [62] on Magma with AFL++ and AFLGo; although Locus intervenes at the fuzzer-feedback layer by instrumenting programs with LLM-synthesized predicates rather than generating seeds, the comparison still isolates the value of SeedSmith's seed-corpus intervention against an alternative LLM-driven approach.

6.2.1 Results on Magma. Table 1 presents trigger times across 23 Magma bugs for all four fuzzers under three seed configurations. SS-only and SS-combined cells are highlighted green when faster than the corresponding Default cell for the same fuzzer.

With SeedSmith seeds, four fuzzers found 22 out of 23 crashes within 24 h, versus 20 with Default seeds. The two additional bugs are `PDF011` and `SQL013`, neither of which Default ever triggers: SeedSmith crashes `PDF011` in one shot and triggers `SQL013` in 23.25 h with AFLRun. Both reflect **C2**, since the bottleneck is satisfying the crash predicates at the sink rather than reaching it (Default reaches `PDF011` in seconds and `SQL013` in ~16 h, but neither satisfies the crash condition within 24 h). Across the 22 SeedSmith-crashed

Table 1: Crash trigger times on Magma: Default vs. SS-only vs. SS-combined seeds. p-value vs. Default in parentheses; green = faster than Default.

Bug ID	AFL++			FairFuzz			AFLGo			AFLRun		
	Default	SS-only	SS-combined	Default	SS-only	SS-combined	Default	SS-only	SS-combined	Default	SS-only	SS-combined
PDF011	T.O.	**1-shot**(<.01)	**1-shot**(<.01)	T.O.	**1-shot**(<.01)	**1-shot**(<.01)	T.O.	**1-shot**(<.01)	**1-shot**(<.01)	T.O.	**1-shot**(<.01)	**1-shot**(<.01)
PDF018	T.O.	**1-shot**(<.01)	**1-shot**(<.01)	T.O.	**1-shot**(<.01)	**1-shot**(<.01)	T.O.	**1-shot**(<.01)	**1-shot**(<.01)	3.25h	**1-shot**(<.01)	**1-shot**(<.01)
PDF021	T.O.	**22.93h**(0.18)	T.O.(1.00)	T.O.	**2.16h**(<.01)	T.O.(1.00)	T.O.	T.O.	T.O.	T.O.	**21.96h**(0.08)	T.O.(1.00)
PHP004	**12.69h**	T.O.(1.00)	14.10h(0.69)	2.33h	T.O.(1.00)	**56.25m**(0.07)	2.28h	T.O.(1.00)	**74.03m**(0.79)	**10.57m**	T.O.(1.00)	13.20m(0.75)
PHP009	5.44h	4.83h(0.03)	**3.17h**(0.52)	T.O.	**17.46h**(0.02)	T.O.(1.00)	25.54m	**2.86m**(<.01)	25.35m(0.38)	49.25m	**5.20m**(0.07)	14.81m(0.16)
PNG001	**22.78h**	23.11h(0.34)	T.O.(0.86)	T.O.	T.O.	T.O.	T.O.	**21.79h**(0.18)	T.O.(1.00)	23.80h	22.07h(0.50)	**21.76h**(0.50)
PNG007	11.51h	19.09h(0.97)	**4.97h**(0.06)	T.O.	**15.25h**(<.01)	21.65h(0.18)	8.46h	7.57h(0.63)	**4.51h**(0.52)	10.37m	10.28m(0.40)	**9.64m**(0.21)
SND017	12.64h	**13.34m**(<.01)	7.58h(0.15)	22.67h	**24.50s**(<.01)	T.O.(0.94)	4.91h	**66.32m**(<.01)	2.37h(0.21)	3.21h	**16.07m**(0.01)	2.74h(0.65)
SND020	**58.16m**	64.59m(0.17)	74.23m(0.63)	8.79h	**33.10m**(0.52)	8.78h(0.45)	30.50m	15.90m(0.69)	**3.63m**(0.19)	**1.65m**	36.33m(0.86)	2.30m(0.76)
SQL002	**53.32m**	2.19h(0.45)	87.92m(0.86)	**21.16h**	T.O.(0.94)	T.O.(0.94)	3.97h	17.31m(<.01)	**14.12m**(<.01)	12.56m	29.57m(0.15)	**4.16m**(<.01)
SQL003	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**17.32h**	T.O.(1.00)	22.52h(0.99)
SQL012	T.O.	T.O.(1.00)	**23.23h**(0.18)	T.O.	T.O.	T.O.	**21.58h**	T.O.(0.94)	T.O.(0.94)	**9.09h**	23.61h(1.00)	11.56h(0.60)
SQL013	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.(1.00)	**23.25h**(0.18)
SQL014	**9.96h**	14.18h(0.79)	12.44h(0.45)	**22.78h**	T.O.(0.86)	T.O.(0.86)	9.70h	4.99h(<.01)	**12.72m**(<.01)	78.36m	**18.48m**(<.01)	25.89m(<.01)
SQL015	23.52h	**19.79h**(0.25)	23.87h(0.56)	T.O.	T.O.	T.O.	**20.31h**	22.41h(0.79)	21.51h(0.61)	21.30h	16.91h(0.35)	**14.42h**(0.08)
SQL020	22.13h	**1-shot**(<.01)	**1-shot**(<.01)	T.O.	**1-shot**(<.01)	**1-shot**(<.01)	21.79h	**1-shot**(<.01)	**1-shot**(<.01)	4.54h	**1-shot**(<.01)	**1-shot**(<.01)
SSL001	22.37h	23.89h(0.95)	**22.29h**(0.91)	T.O.	T.O.	T.O.	**15.78h**	T.O.(0.99)	22.02h(0.96)	3.71h	16.12h(1.00)	**2.09h**(0.15)
SSL020	20.33h	T.O.(0.99)	**19.97h**(0.62)	**19.43h**	T.O.(0.99)	21.81h(0.87)	T.O.	T.O.	T.O.	**5.36h**	T.O.(1.00)	7.96h(0.66)
TIF002	20.84h	**11.81h**(0.02)	11.99h(0.02)	T.O.	**11.59h**(<.01)	14.60h(<.01)	21.00h	**5.50h**(<.01)	20.08h(0.33)	14.32h	37.33m(<.01)	**21.69m**(<.01)
TIF008	22.75h	**3.62h**(<.01)	9.18h(<.01)	21.96h	**5.77h**(<.01)	T.O.(0.86)	T.O.	**9.21h**(<.01)	18.82h(0.02)	17.43h	**2.08m**(<.01)	5.27m(<.01)
TIF014	3.30h	**1-shot**(<.01)	**1-shot**(<.01)	19.62h	**1-shot**(<.01)	**1-shot**(<.01)	42.35m	**1-shot**(<.01)	**1-shot**(<.01)	18.12m	**1-shot**(<.01)	**1-shot**(<.01)
XML003	8.81h	**1-shot**(<.01)	**1-shot**(<.01)	T.O.	**1-shot**(<.01)	**1-shot**(<.01)	3.39h	**1-shot**(<.01)	**1-shot**(<.01)	31.40m	**1-shot**(<.01)	**1-shot**(<.01)
XML009	11.35h	**6.27h**(0.04)	9.67h(0.34)	T.O.	T.O.	T.O.	**31.21m**	10.03h(1.00)	32.40m(0.48)	3.94h	6.79h(0.85)	**30.61m**(0.21)
Geomean Speedup	–	**12.67×**	**11.51×**	–	**21.01×**	**12.29×**	–	**11.14×**	**14.66×**	–	**7.80×**	**13.15×**

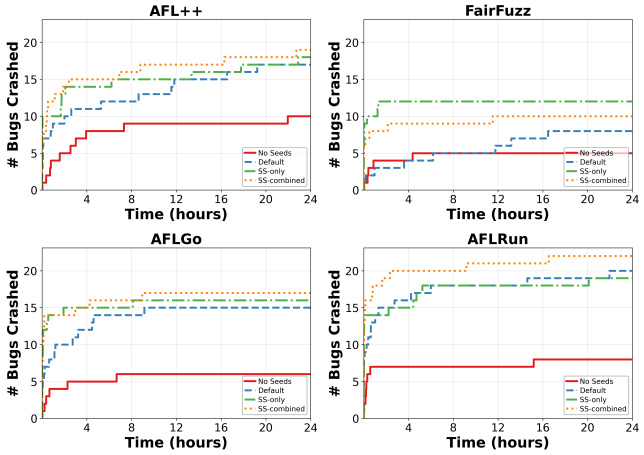

Figure 4: Cumulative unique crashes triggered over 24 h on Magma (23 bugs), one subplot per fuzzer.

bugs, five (`PDF011`, `PDF018`, `SQL020`, `TIF014`, `XML003`) are one-shot crashes where the LLM-generated seed triggers the vulnerability before any fuzzer mutation.

Figure 4 plots cumulative crash counts under all four configurations. The No-seed curve consistently lags the others by 5–10 bugs across all four fuzzers, confirming that initial seed quality is a primary determinant of crash discovery on Magma. The SS-only and SS-combined curves rise steeply within the first hour and then plateau, whereas Default and No-seed climb gradually throughout the 24 h budget; this front-loading reflects one-shot and near-one-shot crashes triggered by LLM-generated seeds at $t \approx 0$ before any meaningful mutation occurs. Notably, AFL++ with SS-combined achieves the second-best performance among the four fuzzers, suggesting that SeedSmith's targeted seed corpus can improve the directed crash-finding capability of a general-purpose fuzzer. This opens the door to a broader range of fuzzers benefiting from SeedSmith seeds, since even non-directed fuzzers can leverage the improved seed quality to trigger deep bugs more efficiently. And SeedSmith can be used as a drop-in seed generator for existing pipeline without requiring integration into the fuzzer's internals, making it widely applicable across fuzzing ecosystems.

SeedSmith is significantly faster than Default ($p < 0.05$) on 10 of the 23 targets for AFL++ and AFLRun, and on 11 for FairFuzz and AFLGo. Beyond the 5 one-shot cases, the remaining accelerated targets see significant time-to-crash reductions through fuzzer-driven mutation of SeedSmith seeds, indicating that even when SeedSmith does not produce a crashing input directly, the seeds encode useful reachability or crash-predicate hints that give the fuzzer a stronger starting point. Geomean speedups under SS-combined range from 11.51× (AFL++) to 14.66× (AFLGo), and under SS-only span 7.80× (AFLRun) to 21.01× (FairFuzz); all four fuzzers show double-digit improvements under at least one configuration.

A finer-grained pattern emerges when we compare SS-only and SS-combined within each fuzzer family. The two coverage-guided fuzzers achieve their best speedup under SS-only (12.67× AFL++ and 21.01× FairFuzz), while the two directed fuzzers peak under SS-combined (14.66× AFLGo and 13.15× AFLRun); the preference is consistent across all four fuzzers. We attribute this to how each fuzzer family uses its initial corpus: coverage-guided fuzzers rely on seeds for both reachability and exploration, so a focused sink-targeted corpus delivers the largest gain, and adding the default

Table 2: Crash trigger times on the Locus subset of Magma.

	AFL++			AFLGo		
Bug	Locus	Default	SS	Locus	Default	SS
PNG007	11.7h	11.5h	**5.0h**	14.1h	8.5h	**4.5h**
SND017	17.8h	12.6h	**7.6h**	14.1h	4.9h	**2.4h**
SQL002	5.7h	**0.9h**	1.5h	1.7h	4.0h	**0.2h**
SQL020	21.7h	22.1h	**1-shot**	22.5h	21.8h	**1-shot**
SSL020	23.6h	20.3h	**20.0h**	T.O.	T.O.	T.O.
TIF014	12.9h	3.3h	**1-shot**	22.5h	0.7h	**1-shot**
XML003	8.3h	8.8h	**1-shot**	2.9h	3.4h	**1-shot**
XML009	**7.9h**	11.4h	9.7h	1.7h	**0.5h**	0.5h

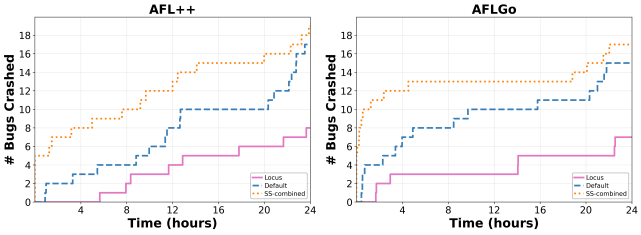

Figure 5: Cumulative unique crashes triggered over 24 h on Magma, SS-combined vs. Locus under AFL++ and AFLGo.

corpus only dilutes the targeting signal. Directed fuzzers, in contrast, need broad coverage for their distance metric to compute meaningful gradients; SS-only alone exercises too few basic blocks for the metric to be informative, while SS-combined provides both the broad coverage that activates the distance metric and the sink-targeted seeds that lead it directly to the bug. SeedSmith therefore narrows the gap between coverage-guided and directed approaches, making the choice of downstream fuzzer less critical to crash-discovery performance, provided the seed configuration is matched to the fuzzer family.

We also compare against Locus [62], which synthesizes predicates that are compiled into the target rather than generating seeds. Across all 23 Magma bugs in 24 hours, SeedSmith triggers 19 crashes on AFL++ and 17 on AFLGo, against Locus's 8 and 7 (Figure 5). On the 8-bug subset where Locus reports results (Table 2), SeedSmith is fastest on 6 of 8 under AFL++ and 6 of 7 valid combinations under AFLGo, with one-shot crashes on `SQL020`, `TIF014`, and `XML003` under both fuzzers.

6.2.2 Results on ARVO. Table 3 reports RMST crash times for the 59 ARVO targets (across 20 projects) where at least one configuration triggered a crash within 24 h. SeedSmith-combined triggers 44 crashes with AFLRun and 38 with AFL++, versus 37 and 26 under Default – a 46% increase on AFL++, considerably larger than the 3 -bug improvement SeedSmith delivers on Magma. We attribute this larger relative gain to ARVO's sparser, less-curated default corpora, which are typical of real-world OSS-Fuzz projects.

Beyond raw counts, SeedSmith unlocks 16 unique crashes across 10 projects that no Default configuration ever triggers: `assimp`, `geos`, `inchi`, `libavc`, `libxaac`, `openjpeg`, `php-src`, `pjsip`, `qpdf`, and `unit`. The diversity of input formats spanned (binary headers, protocol grammars, compression frames) shows that the LLM's pretraining knowledge of format specifications directly supplies the structural invariants that random mutation cannot construct. Figure 6 shows the same patterns sharpened on ARVO: SS-combined plateaus within 4–6 h while Default continues climbing, and the layered gap from No-seed to Default to SeedSmith is wider in absolute bugs.

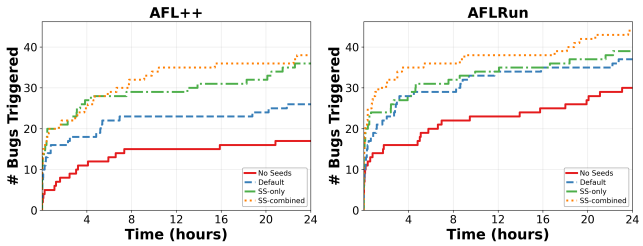

Figure 6: Cumulative unique crashes triggered over 24 h on 115 ARVO targets, one subplot per fuzzer.

Across all 59 ARVO bugs (with timeouts capped at 24 h), SeedSmith seeds yields a 3.09× geomean speedup on AFL++ (sign-test $p = 0.033$) and a 3.02× speedup on AFLRun ($p = 0.043$), both statistically significant at $p < 0.05$. This effect is dominated by the 16 unique unlocks where Default never triggers a crash; on the subset of bugs both configurations trigger, the per-target speedup shrinks to 1.46× on AFL++ and 1.71× on AFLRun and does not reach significance ($p = 0.50$ and $p = 0.18$ at $n = 21$ and 31). SeedSmith's primary contribution on ARVO is therefore *unlocking previously unreachable crashes*, with only modest acceleration on shared bugs. This is consistent with the C2 framing, since for already-reachable crashes both configurations face the same predicate-satisfaction problem at the sink.

6.2.3 Seed Generation Cost. Generating seeds with an LLM incurs additional cost. Table 4 breaks down the cost per project across the 8 Magma projects (per-target measurements are reported in Appendix D). The *Agentic Code Exploration* stage takes on average 118 s and $1.71 per analysis report when using `Claude Sonnet 4`, while each seed costs only $0.0058 and takes 11 s to generate using `gpt-4.1-mini`. On average, a target costs $5.28 and 678 s (~11 min) of wall-clock time, summing to $121.38 across all 23 Magma targets. Compared to a 24 h fuzzing campaign, this upfront investment is negligible: even the most expensive target (`SQL012`, $12.01) costs less than one hour of compute.

6.3 RQ2: Effectiveness of LLM Reasoning

We answer **RQ2** through a manual investigation of SeedSmith's seed-generation process. We use Magma `PDF018` as a case study (Section 6.3.1) and then summarize patterns observed across the other accelerated targets (Section 6.3.2).

6.3.1 Case Study. We use Magma `PDF018`, a null-pointer dereference in the `poppler` PDF rendering library, as a case study to illustrate how SeedSmith's LLM-driven analysis addresses both **C1** and **C2**. As Figure 7 shows, the crash occurs in `AnnotInk::draw` (Line 18–19) and is reached through a virtual dispatch from `Page::displaySlice` (Line 3).

Table 3: Crash trigger times on ARVO: SeedSmith vs. Default seeds, only targets crashed at least once are shown.

		AFL++		AFLRun				AFL++		AFLRun				AFL++		AFLRun	
Proj.	**ID**	Def.	SS	Def.	SS	**Proj.**	**ID**	Def.	SS	Def.	SS	**Proj.**	**ID**	Def.	SS	Def.	SS
aom	42533504	**17.14h**	19.55h	12.54h	**10.90h**	libavc	42531113	T.O.	T.O.	T.O.	**23.67h**	libxml2	42532747	T.O.	T.O.	**21.87h**	T.O.
	42533540	**15.13h**	16.92h	12.33h	**10.29h**		42531141	T.O.	T.O.	**22.42h**	T.O.		42533922	**19.95h**	20.34h	**20.79h**	21.65h
assimp	42528235	T.O.	**1-shot**	T.O.	**1-shot**	libdwarf	42528680	16.25m	**11.51m**	**17.30m**	18.47m		42533950	22.83h	**22.58h**	22.05h	**21.57h**
geos	42532863	T.O.	**21.74h**	T.O.	**23.53h**		42528792	22.23m	**10.42m**	50.64m	**50.54m**		42534044	**17.04h**	20.37h	T.O.	T.O.
inchi	42534536	T.O.	**19.64h**	T.O.	**6.68h**		42528883	6.17m	**5.83m**	32.77m	**16.21m**		42534845	12.99h	**7.71h**	10.16h	**2.75h**
	42534760	T.O.	**1-shot**	3.19h	**1-shot**	libical	42536107	**4.18h**	16.62h	**11.45h**	13.53h		42537493	1.52m	**49.42s**	39.27s	**37.25s**
	42534959	T.O.	T.O.	23.88h	**22.13h**	libssh2	42531314	T.O.	T.O.	**5.60h**	7.62h	openjpeg	42535258	T.O.	**6.32m**	T.O.	T.O.
	42535235	T.O.	T.O.	T.O.	**23.48h**	libtpms	42537128	T.O.	**22.67h**	17.99h	**17.51h**	php-src	42529542	**22.12h**	T.O.	T.O.	T.O.
	42536064	T.O.	**1.38m**	26.97m	**8.07m**	libxaac	42527401	T.O.	**18.13h**	T.O.	T.O.		42529650	T.O.	**1-shot**	T.O.	**1-shot**
	42536079	T.O.	**22.64h**	**14.01h**	T.O.		42527409	T.O.	T.O.	**22.22h**	T.O.	pjsip	42535147	T.O.	**1-shot**	T.O.	**1-shot**
	42536250	T.O.	**22.28h**	T.O.	T.O.		42527540	T.O.	T.O.	**23.18h**	T.O.	qpdf	42531940	T.O.	T.O.	**22.77h**	T.O.
	42536593	T.O.	T.O.	T.O.	**22.46h**		42527636	T.O.	**23.88h**	T.O.	T.O.		42535152	T.O.	**21.75h**	T.O.	**21.90h**
	42536641	T.O.	**22.38h**	**22.46h**	23.16h		42531547	**1.91h**	2.82h	**7.57h**	18.02h	simdjson	42534862	T.O.	T.O.	**2.40h**	2.40h
lcms	42529812	**19.89h**	T.O.	T.O.	T.O.	libxml2	42527008	16.76m	**1-shot**	28.20m	**1-shot**		42534891	T.O.	T.O.	**1.00s**	12.58s
	42529915	1.34h	**35.28m**	59.71m	**40.66s**		42528997	**17.92h**	T.O.	**15.19h**	16.19h		42534894	T.O.	T.O.	**1.68s**	11.43s
	42529931	5.18h	**2.89h**	2.67h	**17.75m**		42531092	**23.26h**	T.O.	**21.89h**	23.41h		42534941	T.O.	T.O.	**35.29s**	2.40h
	42530151	18.86h	**16.27h**	15.74h	**8.43h**		42531126	T.O.	**21.63h**	21.90h	**19.54h**	unit	42536363	T.O.	**15.98h**	T.O.	**1.99h**
libavc	42530446	T.O.	T.O.	T.O.	**23.96h**		42531203	**12.01h**	20.75h	21.14h	**15.87h**	xz	42534913	1.16s	**1-shot**	T.O.	**1-shot**
	42530451	T.O.	T.O.	T.O.	**22.50h**		42531481	**23.76h**	23.85h	T.O.	T.O.	zstd	42532756	**20.66h**	22.06h	23.82h	**21.63h**
	42530453	**22.14h**	T.O.	T.O.	T.O.		42531532	**14.47h**	16.21h	**17.11h**	19.76h						

Table 4: Cost and time analysis per project. Counts in parentheses denote the number of targets per project. Rpt. = Analysis report, Seed = Generated seed.

Project	Total		Avg. Rpt.		Avg. Seed	
	Cost ($)	Time (s)	Cost ($)	Time (s)	Cost ($)	Time (s)
libpng (2)	9.25	1373	1.47	108	0.0076	13
libsndfile (2)	9.85	1410	1.60	135	0.0046	10
libtiff (3)	16.38	2468	1.73	131	0.0083	15
libxml2 (2)	10.02	1161	1.62	122	0.0052	7
openssl (2)	8.45	1132	1.35	109	0.0060	8
php (2)	8.40	1801	1.33	131	0.0074	17
poppler (3)	10.37	1898	1.09	104	0.0067	11
sqlite3 (7)	48.66	4357	2.28	116	0.0037	9
Overall (23)	**121.38**	**15600**	**1.71**	**118**	**0.0058**	**11**

When an annotation has `/Subtype /Ink`, an `AnnotInk` object is instantiated (Lines 4–7) and its `parseInkList` method (Lines 8–15) is called to parse the `/InkList` array. `parseInkList` zero-initializes the `AnnotPath*` array via `memset` and only populates entries whose corresponding PDF element is an array; non-array elements (e.g., `null`, integers) leave the slot as `NULL`. `AnnotInk::draw` (Lines 16–21) then iterates over this array and dereferences `path->getCoordsLength()` at Line 19 without a null check, resulting in a crash.

This bug exemplifies both challenges. For **C1**, static analysis cannot resolve which concrete `draw()` override is invoked through the virtual dispatch at Line 3, so directed fuzzers relying on call-graph distance have no guidance toward `AnnotInk::draw`. For **C2**, constructing the malformed `/InkList` requires PDF syntax and `parseInkList` semantics that random mutation cannot discover, but that the LLM draws from its pretraining on format specifications.

Reasoning trace. Figure 8 traces the agent's reasoning. On `PDF018`, CodeQL fails to build a database, so the agent receives only *I1* (the harness `LLVMFuzzerTestOneInput`) and *I2* (the sink `AnnotInk::draw`); *I3* is empty. Reading the sink, the agent

```
1  // Page.cc -- Page::displaySlice
2  annotList = getAnnots();
3  annotList->getAnnot(i)->draw(gfx, printing); // indirect virtual
       call
4  // Annot.cc -- Annots::createAnnot
5  } else if (!strcmp(typeName, "Ink")) {
6      annot = new AnnotInk(doc, std::move(dictObject), obj);
7  }
8  // Annot.cc -- AnnotInk::parseInkList
9  void AnnotInk::parseInkList(Array *array) {
10     for (int i = 0; i < inkListLength; i++) {
11         Object obj2 = array->get(i);
12         if (obj2.isArray()) {
13             inkList[i] = new AnnotPath(obj2.getArray());
14     }
15 }
16 // Annot.cc -- AnnotInk::draw
17 for (int i = 0; i < inkListLength; ++i) {
18     const AnnotPath *path = inkList[i];       // NULL
19     if (path->getCoordsLength() != 0) {       // CRASH: null
         pointer dereference
20     }
21 }
22
```

Figure 7: Simplified `PDF018` trigger chain: from `Page::displaySlice` through `AnnotInk::parseInkList` to the null-pointer dereference in `AnnotInk::draw`.

observes that `AnnotInk::draw` iterates over `inkList` and dereferences `path->getCoordsLength()` with no null check, and asks when `inkList[i]` can be null (*Q1–Q3*). *Q2*'s retrieval of `parseInkList` exposes the critical pattern: `memset(inkList, 0, ...)` zero-initializes the array, then only populates entries where `obj2.isArray()` holds, so any non-array PDF element silently leaves a null slot. Reading the harness, *Q4–Q6* reconstruct the call chain `LLVMFuzzerTestOneInput` → `render_page` → `displayPageSlice` → `Page::displaySlice` → `AnnotInk::draw`. Combining the two findings, the agent emits a

PDF whose `/InkList` contains a valid coordinate array followed by integer 42: 42 passes the PDF lexer and the array-of-arrays validator (both accept arbitrary objects), but fails `parseInkList`'s `isArray` check, leaving `inkList[1]` null. `AnnotInk::draw` dereferences it on the first iteration, producing the one-shot crash recorded in Table 1. What this trace surfaces is an interprocedural invariant (zeroed slot → null deref) revealed only by reading `parseInkList` and `draw` together; neither static call-graph distance nor a one-shot prompt over the sink alone would expose it.

Inputs. *I1* `LLVMFuzzerTestOneInput` (harness) *I2* `AnnotInk::draw` (sink) *I3* ∅ *(CodeQL fails on* `poppler`*)*

Sink-side queries *(when is* `inkList[i]` *null?):*
- Q1. search `AnnotInk::` → sibling methods (`parseInkList`, `draw`)
- Q2. search `parseInkList` → `memset(0)`; only array entries populated
- Q3. search `getCoordsLength` → no null guard before deref

Harness-side queries *(how is sink reached?):*
- Q4. search `render_page` → `displayPageSlice()`
- Q5. search `displayPageSlice` → `Page::displaySlice`
- Q6. search `Page::displaySlice` → `annot->draw()`

Synthesized seed. `/InkList [[100 150 150 150 200 100], 42]`
Integer 42 fails `isArray()`, leaving `inkList[1]` null at draw time.

Figure 8: Reasoning trace for `PDF018`: starting from raw harness and sink, the agent's six tool-driven queries fan out into sink-side and harness-side sub-goals and converge on a single crash-triggering seed.

6.3.2 Analysis of Accelerated Magma Targets. The pattern observed in `PDF018` generalizes across the other nine targets highlighted in green in Table 1 for *every* fuzzer (`PDF011`, `PHP009`, `PNG007`, `SND017`, `SQL020`, `TIF002`, `TIF008`, `TIF014`, `XML003`). All nine require format-specific structural invariants (**C2**) that random mutation from generic corpora is difficult to construct: SQLite tagged-union (`SQL020`), JPEG/EXIF MakerNote vendor-prefix (`PHP009`), XML external-PE default-option (`XML003`), PDF xref negative-index (`PDF011`), WAV `WAVE_FORMAT_EXTENSIBLE` channel-mask (`SND017`), and so on. Five of the nine (`TIF002`, `TIF008`, `TIF014`, `XML003`, `PNG007`) also hide the sink behind function pointers or virtual dispatch (**C1**): libtiff's codec-dispatch tables, libxml2's SAX callbacks, and libpng's warning/error handlers are populated at runtime and invisible to static call-graph construction. Across all nine targets, the analysis agent reads the relevant format and dispatch code together (as in the `PDF018` trace above) and produces seeds that exercise the correct dispatch chain: directly triggering the crash for the five one-shot cases, and seeding the fuzzer within mutation-reachable distance for the rest.

6.4 RQ3: Ablation Study

To address **RQ3**, we ablate SeedSmith's three components on Magma. The LLM seed generator's contribution is established in RQ1 against Default seeds; here we isolate two further design choices: (i)The *scan strategy* is the search tool's context-aware retrieval rules, which return semantically complete units (function body, type declaration, or configuration block) instead of raw grep matches (Section 4.2); we ablate it by switching the tool back to raw grep output. (ii)The *control-flow support* is the CodeQL-derived, path-optimized harness-to-sink path supplied as input *I3* (Section 4.1); we ablate it by replacing *I3* with an empty path. We compare full SeedSmith against both ablated variants on 8 Magma bugs. CodeQL's database extraction succeeds on only these 8 targets due to build-system incompatibilities common in real-world C/C++ projects [29]; this is an external limitation of CodeQL, and on the other 15 targets SeedSmith gracefully falls back to the no-CodeQL configuration that RQ1 already evaluates. The 8-bug subset still spans six input formats (PDF, PNG, SND, SSL, TIFF, XML), preserving Magma's format diversity. The speedups reported below are *marginal* contributions of each component on top of the LLM seed generator's RQ1 baseline, not absolute speedups over Default. Concretely, each ratio is computed as $T_{\text{w/o component}}/T_{SeedSmith}$ on the same bug-fuzzer pair, isolating how much that component accelerates SeedSmith relative to a version of SeedSmith without it; the speedup of SeedSmith over Default itself is the RQ1 number.

Table 5: SeedSmith seeds vs. SeedSmith seeds w/o Scan Strategy.

	AFL++		FairFuzz		AFLGo		AFLRun	
Bug	w/o Scan	SS	w/o Scan	SS	w/o Scan	SS	w/o Scan	SS
PDF011	T.O.	**1-shot**	T.O.	**1-shot**	T.O.	**1-shot**	T.O.	**1-shot**
PNG001	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	23.4h	**21.8h**
PNG007	7.6h	**5.0h**	T.O.	**21.6h**	**4.3h**	4.5h	**7.1m**	9.6m
SND017	**1-shot**	7.6h	**1-shot**	T.O.	**1-shot**	2.4h	**1-shot**	2.7h
SND020	**47.3m**	1.2h	9.6h	**8.8h**	24.0m	**3.6m**	**2.1m**	2.3m
SSL001	**21.8h**	22.3h	T.O.	T.O.	T.O.	**22.0h**	5.6h	**2.1h**
TIF014	13.2m	**1-shot**	10.0h	**1-shot**	35s	**1-shot**	24s	**1-shot**
XML003	**1-shot**	**1-shot**	**1-shot**	**1-shot**	**1-shot**	**1-shot**	**1-shot**	**1-shot**
Geomean Speedup	**3.0×**		**5.9×**		**3.0×**		**2.1×**	

Table 6: SeedSmith seeds vs. SeedSmith seeds w/o CodeQL-based control-flow support.

	AFL++		FairFuzz		AFLGo		AFLRun	
Bug	w/o CodeQL	SS	w/o CodeQL	SS	w/o CodeQL	SS	w/o CodeQL	SS
PDF011	T.O.	**1-shot**	T.O.	**1-shot**	T.O.	**1-shot**	T.O.	**1-shot**
PNG001	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	24.0h	**21.8h**
PNG007	9.6h	**5.0h**	T.O.	**21.6h**	5.0h	**4.5h**	21.1m	**9.6m**
SND017	11.9h	**7.6h**	T.O.	T.O.	6.4h	**2.4h**	3.0h	**2.7h**
SND020	1.5h	**1.2h**	9.1h	**8.8h**	21.6m	**3.6m**	3.9m	**2.3m**
SSL001	**20.3h**	22.3h	T.O.	T.O.	**15.3h**	22.0h	9.1h	**2.1h**
TIF014	5.4m	**1-shot**	1.5h	**1-shot**	1.7m	**1-shot**	28s	**1-shot**
XML003	4.0m	**1-shot**	T.O.	**1-shot**	T.O.	**1-shot**	27s	**1-shot**
Geomean Speedup	**30.2×**		**540.9×**		**71.4×**		**13.8×**	

Effect of the Scan Strategy. The scan strategy yields per-fuzzer additional geomean speedups of 2.1× to 5.9× (Table 5). Its value is most apparent on targets with multiple plausible execution paths, where directing the agent toward the right one avoids exploratory overhead. On `TIF014`, SeedSmith achieves one shot for all four fuzzers while SeedSmith w/o Scan Strategy requires 24 s–13.2 m across the directed/coverage fuzzers and up to 10.0 h for FairFuzz; on `PDF011`, `PNG001`, and `SSL001`, SeedSmith either uniquely triggers the crash or matches the ablated variant. `SND017` is the principal exception: SeedSmith w/o Scan Strategy achieves one-shot across all fuzzers while SeedSmith requires 2.4 h–7.6 h, because the vulnerability is directly reachable through a single call chain and the scan strategy's path exploration introduces unnecessary indirection.

Effect of Control Flow Support. Having an initial call graph to refine rather than reconstruct from scratch substantially reduces the exploration burden. The component yields per-fuzzer additional geomean speedups of 30.2× on AFL++, 71.4× on AFLGo, and 13.8× on AFLRun (Table 6), driven primarily by `PDF011` and `XML003`, where the ablated variant times out while SeedSmith crashes in one shot. The spread reflects how much each fuzzer relies on a complete static call graph: AFLGo's distance metric depends entirely on it, so missing indirect edges blinds its scheduling, while AFLRun augments the call graph at runtime as new edges are observed, partially compensating for the missing edges when CodeQL is unavailable. FairFuzz is reported separately at 540.9×, but this number is not directly comparable: without control-flow support FairFuzz fails to trigger most bugs within 24 h, so only `TIF014` and `XML003` contribute non-T.O. ratios, dominating the geometric mean. The main exception is `SSL001`, on which the ablated variant is faster for AFL++ and AFLGo; the SSL vulnerability hinges on a protocol state-machine error driven by input semantics, so extra call-graph information does not help.

Summary. The two components contribute incrementally and non-redundantly: the scan strategy improves the quality of information the agent receives per code-search query, while control-flow support gives the agent an initial call graph to refine rather than reconstruct from scratch. Both yield positive marginal speedups across all four fuzzers (with the SND017/SSL001 exceptions noted above), and their combination produces the strongest SeedSmith variant evaluated in RQ1.

7 Discussion and Limitations

LLM Seed Generation. SeedSmith's ability to generate crashing test cases highly depends on the LLM's code-understanding ability. When LLMs fail to correctly identify the root cause of the vulnerability in the sink function or misdiagnose the trigger condition, the probability of generating a crashing input drops. This can be mitigated by fine-tuning a model with enhanced vulnerability-analysis capability. This dependency is also an advantage: as LLMs continue to improve at code understanding and vulnerability reasoning, SeedSmith's seed-generation quality improves correspondingly without any changes to the system itself.

Precision of the Sink Function. In our evaluation, we provide the first function on the crashing stack trace to SeedSmith as the sink function. However, the root cause of the vulnerability might not be in the provided function. The real vulnerability might be located in the other functions of the stack trace, or it might not be on the stack trace at all. By merely targeting the provided sink function without providing any context or information about the actual vulnerability, the LLM is less ineffective. For SeedSmith, this can be mitigated by providing additional information to the LLM. For example, for crash reproduction, giving to the LLM the original crash report can make the LLM aware of the possibility that the vulnerability might exist in other functions on the stack trace, thus rendering it easier for LLM to locate the root cause of the vulnerability by potentially exploring more functions on the stack trace.

Zero-Day Discovery. Our evaluation targets the N-day reproduction setting, where the sink function is given as input to SeedSmith. SeedSmith does not perform sink localization on its own, so applying it to zero-day discovery requires pairing it with a sink-discovery component, such as static bug-pattern detection [22] or learned vulnerability symptoms [38]. We have evaluated SeedSmith as a separate component in our AIxCC finals submission, paired with a customized setup for confirming the sink function. In that deployment, SeedSmith seeds independently one-shot triggered 20 different targets, and further helped downstream fuzzers reach seven zero-day vulnerabilities, four of which were not discovered by any other team. This demonstrates that SeedSmith generalizes to zero-day discovery when paired with a sink-discovery component.

8 Related Work

Directed Fuzzing. Directed fuzzing aims to improve the efficiency of vulnerability discovery by steering the fuzzer toward specific, potentially vulnerable code regions. These target regions may include code that performs memory operations [22], calls to known vulnerable functions [47], code matching learned bug symptoms [38], recently patched code [36], or newly committed changes [51]. By narrowing the focus, directed fuzzing enhances the likelihood of uncovering security-critical bugs in a more targeted and resource-efficient manner.

Despite variations in how target locations are determined [22, 36, 42, 47, 51], directed fuzzers generally share the goal of prioritizing program paths that are semantically or syntactically closer to the targets. Prior work in this area can be broadly categorized into three groups based on the stage of the fuzzing pipeline they aim to optimize: seed scheduling, seed mutation, and mutation tracing.

Among these, seed scheduling has received significant attention. AFLGo [10] pioneered the approach of assessing seeds based on the distance between their execution traces and the target locations at the basic block level, prioritizing seeds with shorter distances for mutation. Subsequent works refined this distance metric using more precise basic block–level granularity [17, 26], function-level abstractions [12], data distances [27], and even inter-target correlations [25, 30, 60], all aiming to guide the fuzzing process more effectively toward the target locations.

Seed mutation strategies have also been explored to enhance the likelihood that generated inputs reach target code regions. FairFuzz [28] introduces a mutation mask that biases mutations toward input bytes associated with rare branches, thereby increasing the probability of exercising hard-to-reach paths. RDFuzz [54] further improves this by identifying and preserving input content that is sensitive to distance metrics, generating mutations more likely to minimize the distance to targets. In parallel, symbolic execution tools [11, 21, 43, 55] can generate inputs that precisely satisfy the constraints of a path leading to a target. However, symbolic execution suffers from poor scalability when the target is deeply embedded in the program state space, as it must first enumerate or search a vast number of paths to isolate one that reaches the goal [7, 34].

Finally, prior work on optimizing the mutation tracing stage focuses on evaluating whether generated inputs are likely to reach

the target and pruning those that are not. FuzzGuard [63] trains a neural network to predict target reachability and filter out unpromising inputs before execution. Beacon [24] uses lightweight static analysis to infer input preconditions and discard infeasible mutations.

LLM-Augmented Fuzzing. Recent advances in LLMs show that they can synthesize high-quality structured text, which can be used to enhance fuzzing performance from multiple aspects. The most straightforward usage of LLMs is seed generation. Several prior works [15, 16, 19, 50] leveraged the LLM's superior code generation capability to produce fuzzing inputs for programming language compilers, interpreters, and library APIs. LLMs have also been used to directly generate commands or inputs to popular applications as an initial seed corpus [6]. Magneto [61] took a step further by feeding the LLM fine-grained program structural information retrieved through static analysis to help it generate initial inputs for directed fuzzing of Java dependency libraries.

Although LLMs excel at generating textual inputs, they are less effective when targets consume non-textual inputs. Instead of directly using the LLM's outputs as fuzzing seeds, G2FUZZ [58] instructs the LLM to emit Python scripts that act as input generators and mutators; ProphetFuzz [48] drives an LLM-based configuration fuzzer that explores option combinations. and ChatAFL [39] uses an LLM to extract protocol grammar from RFC-style documentation. Orthogonal to fuzzing input generation, LLMs also excel in fuzzer driver generation [31, 33, 57]. LLMs are pre-trained on open-source projects, which makes them good at generating fuzzer drivers that can explore uncovered library code.

9 Conclusion

In this paper, we present SeedSmith, a system designed to enhance vulnerability discovery in code regions surrounding sink functions. We demonstrate that SeedSmith effectively mitigates the "cold-start" problem of modern fuzzers, significantly reducing both time-to-reach and time-to-crash.

We evaluated SeedSmith on 23 Magma bugs and 115 ARVO challenges across 26 projects. In the Magma benchmarks, the generated seeds allowed fuzzers to identify 22 of the 23 bugs. This outperformed default seeds, which triggered 20 bugs. Additionally, the system delivered significant geometric mean speedups, specifically 11.51× for AFL++ and 14.66× for AFLGo. On ARVO, fuzzers using SeedSmith seeds trigger 16 bugs that AFLRun and AFL++ with default seeds never trigger, spanning 10 projects with diverse input formats. While the time-to-crash speedup on bugs that both configurations trigger is not statistically significant ($p = 0.58$), the primary advantage of SeedSmith lies in expanding the set of reachable crashes, enabling fuzzers to trigger vulnerabilities that mutation alone cannot reach within the time budget.

References

[1] 2025. Artificial Intelligence Cyber Challenge. https://aicyberchallenge.com/.

[2] 2025. Neo4j Graph Database & Analytics – The Leader in Graph Databases. https://neo4j.com/

[3] 2026. CodeQL. https://codeql.github.com/.

[4] 2026. Langchain. https://www.langchain.com/.

[5] 2026. tree-sitter. https://tree-sitter.github.io/tree-sitter/.

[6] Asmita, Yaroslav Oliinyk, Michael Scott, Ryan Tsang, Chongzhou Fang, and Houman Homayoun. 2024. Fuzzing BusyBox: Leveraging LLM and Crash Reuse for Embedded Bug Unearthing. 883–900. https://www.usenix.org/conference/usenixsecurity24/presentation/asmita

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[8] Andrew Bao, Wenjia Zhao, Yanhao Wang, Yueqiang Cheng, Stephen McCamant, and Pen-Chung Yew. 2025. From Alarms to Real Bugs: Multi-target Multi-step Directed Greybox Fuzzing for Static Analysis Result Verification. 6977–6997. https://www.usenix.org/conference/usenixsecurity25/presentation/bao-andrew

[9] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1985–2002. https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2329–2344.

[11] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Vol. 8. 209–224.

[12] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2095–2108.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[14] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. doi:10.1109/SP.2018.00046

[15] 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 *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 423–435. doi:10.1145/3597926.3598067

[16] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3597503.3623343

[17] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. Windranger: A directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the International Conference on Software Engineering*. 2440–2451.

[18] Rafael Dutra, Rahul Gopinath, and Andreas Zeller. 2023. FormatFuzzer: Effective Fuzzing of Binary File Formats. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 53 (Dec. 2023), 29 pages. doi:10.1145/3628157

[19] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1656–1668. doi:10.1145/3650212.3680389

[20] Elia Geretto, Andrea Jemmett, Cristiano Giuffrida, and Herbert Bos. 2025. LibAFLGo: Evaluating and Advancing Directed Greybox Fuzzing. In *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*. 355–373. doi:10.1109/EuroSP63326.2025.00029 ISSN: 2995-1356.

[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*. 213–223.

[22] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for {Overflows}: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the USENIX Security Symposium*. 49–64.

[23] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.

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

[25] Heqing Huang, Peisen Yao, Hung-Chun Chiu, Yiyuan Guo, and Charles Zhang. 2024. Titan: Efficient multi-target directed greybox fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1849–1864.

[26] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. {DAFL}: Directed Grey-box Fuzzing guided by Data Dependency. In *Proceedings of the USENIX Security Symposium*. 4931–4948.

[27] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided directed greybox fuzzing. In *Proceedings of the USENIX Security Symposium*. 3559–3576.

[28] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 475–485.

[29] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. doi:10.48550/arXiv.2405.17238 arXiv:2405.17238 [cs].

[30] Hongliang Liang, Xinglin Yu, Xianglin Cheng, Jie Liu, and Jin Li. 2023. Multiple targets directed greybox fuzzing. *IEEE Transactions on Dependable and Secure Computing* 21, 1 (2023), 325–339.

[31] Dongge Liu, Oliver Chang, Jonathan metzman, Martin Sablotny, and Mihai Maruseac. 2024. OSS-Fuzz-Gen: Automated Fuzz Target Generation. https://github.com/google/oss-fuzz-gen original-date: 2024-01-25T00:51:49Z.

[32] Danushka Liyanage, Marcel Böhme, Chakkrit Tantithamthavorn, and Stephan Lipp. 2023. Reachable Coverage: Estimating Saturation in Fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 371–383. doi:10.1109/ICSE48619.2023.00042 ISSN: 1558-1225.

[33] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt Fuzzing for Fuzz Driver Generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. Association for Computing Machinery, New York, NY, USA, 3793–3807. doi:10.1145/3658644.3670396

[34] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. 2011. Directed symbolic execution. In *International Static Analysis Symposium*. Springer, 95–111.

[35] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. arXiv:2303.17651 [cs.CL] https://arxiv.org/abs/2303.17651

[36] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 235–245.

[37] Xiang Mei, Pulkit Singh Singaria, Jordi Del Castillo, Haoran Xi, Abdelouahab, Benchikh, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, Hammond Pearce, and Brendan Dolan-Gavitt. 2024. ARVO: Atlas of Reproducible Vulnerabilities for Open Source Software. doi:10.48550/arXiv.2408.02153 arXiv:2408.02153 [cs].

[38] Dongyu Meng, Michele Guerriero, Aravind Machiry, Hojjat Aghakhani, Priyanka Bose, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. 2021. Bran: Reduce Vulnerability Search Space in Large Open Source Repositories by Learning Bug Symptoms. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (Virtual Event, Hong Kong) *(ASIA CCS '21)*. Association for Computing Machinery, New York, NY, USA, 731–743. doi:10.1145/3433210.3453115

[39] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large Language Model guided Protocol Fuzzing. In *Network and Distributed System Security (NDSS) Symposium 2024*. San Diego, CA, USA. https://www.ndss-symposium.org/ndss-paper/large-language-model-guided-protocol-fuzzing/

[40] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3597503.3639187

[41] nginx. [n. d.]. nginx is the world's most popular Web Server. https://github.com/nginx/nginx.

[42] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. {ParmeSan}: Sanitizer-guided greybox fuzzing. In *Proceedings of the USENIX Security Symposium*. 2289–2306.

[43] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with {SymCC}: Don't interpret, compile!. In *Proceedings of the USENIX Security Symposium*. 181–198.

[44] Huanyao Rong, Wei You, XiaoFeng Wang, and Tianhao Mao. 2024. Toward Unbiased Multiple-Target Fuzzing with Path Diversity. 2475–2492. https://www.usenix.org/conference/usenixsecurity24/presentation/rong

[45] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC '16)*. Association for Computing Machinery, New York, NY, USA, 265–266. doi:10.1145/2892208.2892235

[46] Chenlin Wang, Wei Meng, Changhua Luo, and Penghui Li. 2025. Predator: Directed Web Application Fuzzing for Efficient Vulnerability Validation. In *2025 IEEE Symposium on Security and Privacy (SP)*. 886–902. doi:10.1109/SP61157.2025.00066 ISSN: 2375-1207.

[47] Chenlin Wang, Wei Meng, Changhua Luo, and Penghui Li. 2025. Predator: Directed Web Application Fuzzing for Efficient Vulnerability Validation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 886–902.

[48] Dawei Wang, Geng Zhou, Li Chen, Dan Li, and Yukai Miao. 2024. ProphetFuzz: Fully Automated Prediction and Fuzzing of High-Risk Option Combinations with Only Documentation via Large Language Model. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. Association for Computing Machinery, New York, NY, USA, 735–749. doi:10.1145/3658644.3690231

[49] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 579–594. doi:10.1109/SP.2017.23

[50] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the International Conference on Software Engineering*. 1–13.

[51] Yi Xiang, Xuhong Zhang, Peiyu Liu, Shouling Ji, Hong Liang, Jiacheng Xu, and Wenhai Wang. 2024. Critical code guided directed greybox fuzzing for commits. In *Proceedings of the USENIX Security Symposium*. 2459–2474.

[52] Yijiang Xu, Hongrui Jia, Liguo Chen, Xin Wang, Zhengran Zeng, Yidong Wang, Qing Gao, Jindong Wang, Wei Ye, Shikun Zhang, et al. 2024. ISC4DGF: Enhancing Directed Grey-Box Fuzzing with LLM-Driven Initial Seed Corpus Generation. *arXiv preprint arXiv:2409.14329* (2024).

[53] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] https://arxiv.org/abs/2210.03629

[54] Jiaxi Ye, Ruilin Li, and Bin Zhang. 2020. RDFuzz: Accelerating directed fuzzing with intertwined schedule and optimized mutation. *Mathematical Problems in Engineering* 2020, 1 (2020), 7698916.

[55] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the USENIX Security Symposium*. 745–761.

[56] Cen Zhang, Younggi Park, Fabian Fleischer, Yu-Fu Fu, Jiho Kim, Dongkwan Kim, Youngjoon Kim, Qingxiao Xu, Andrew Chin, Ze Sheng, Hanqing Zhao, Brian J. Lee, Joshua Wang, Michael Pelican, David J. Musliner, Jeff Huang, Jon Silliman, Mikel Mcdaniel, Jefferson Casavant, Isaac Goldthwaite, Nicholas Vidovich, Matthew Lehman, and Taesoo Kim. 2026. SoK: DARPA's AI Cyber Challenge (AIxCC): Competition Design, Architectures, and Lessons Learned. doi:10.48550/arXiv.2602.07666 arXiv:2602.07666 [cs].

[57] 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 *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1223–1235. doi:10.1145/3650212.3680355

[58] Kunpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. 2025. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. 6999–7018. https://www.usenix.org/conference/usenixsecurity25/presentation/zhang-kunpeng

[59] Yujian Zhang, Yaokun Liu, Jinyu Xu, and Yanhao Wang. 2024. Predecessor-aware Directed Greybox Fuzzing. IEEE Computer Society, 1884–1900. doi:10.1109/SP54263.2024.00040

[60] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2023. {FISHFUZZ}: Catch deeper bugs by throwing larger nets. In *Proceedings of the USENIX Security Symposium*. 1343–1360.

[61] Zhuotong Zhou, Yongzhuo Yang, Susheng Wu, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. Magneto: A Step-Wise Approach to Exploit Vulnerabilities in Dependent Libraries via LLM-Empowered Directed Fuzzing. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering*. 1633–1644.

[62] Jie Zhu, Chihao Shen, Ziyang Li, Jiahao Yu, Yizheng Chen, and Kexin Pei. 2025. Locus: Agentic Predicate Synthesis for Directed Fuzzing. doi:10.1145/3744916.3773102 arXiv:2508.21302 [cs].

[63] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. {FuzzGuard}: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *Proceedings of the USENIX Security Symposium*. 2255–2269.

A Open Science

We support the ACM CCS 2026 Open Science policy and enumerate below the artifacts needed to evaluate SeedSmith's core contributions. The artifact submitted along with the paper is available at https://anonymous.4open.science/r/SeedSmith-47CF.

- **SeedSmith implementation.** The full source of the SeedSmith agentic pipeline, including the Analysis Agent, the Seed Generation Agent, the LLM-facing call-graph and indexing utilities, and the end-to-end driver scripts that build clang indexes, CodeQL databases, and instrumented OSS-Fuzz artifacts for each target.
- **Benchmarks.** The list of Magma bugs (23 targets) and ARVO challenges (115 targets across 26 projects) used in the evaluation, along with the exact commit/build configurations and sink specifications. The Magma benchmark we run is available at https://github.com/vusec/magma-directed. The ARVO dataset we use is available at https://github.com/n132/ARVO-Meta.
- **Baselines.** AFLRun code is available at https://zenodo.org/records/12797635, and Locus code at https://github.com/jiezhuzzz/Locus. The remaining baselines' code is available at https://github.com/vusec/magma-directed.
- **Fuzzing driver scripts.** We ran all fuzzing experiments on a Kubernetes cluster. The Kubernetes configuration files are omitted from the submitted artifact because they contain references to a private Docker registry and private GitHub repositories that would deanonymize the authors. These scripts, the private Docker images, and repositories will be made public upon acceptance of the paper. The core fuzzing harnesses and benchmarks presented in the paper are all open source can be run locally without Kubernetes.
- **Generated seed corpora.** The LLM-generated seeds were produced for each target, so that reviewers can reproduce fuzzing runs without re-invoking the LLM backends.
- **Raw experiment results.** The per-trial trigger times that back every figure and table in the paper, with one file per benchmark and ablation. The column schema (10 trials per cell, 24-hour campaigns) and the one-shot-bug list per configuration are documented alongside the data files.

B Ethics Considerations

In this work, we present SeedSmith, a tool that can be used for vulnerability discovery. As our work operates on the ARVO dataset, a curated benchmark of known vulnerabilities, no new bugs have been uncovered and, thus, no vulnerability disclosure was necessary.

C Generative AI usage

In our work, LLMs were used for editorial purposes in this manuscript, and the authors inspected all outputs to ensure accuracy and originality.

We used Claude Code to generate utility scripts to run fuzzing campaigns for baselines and processing raw experiment data. All the running results and experiment data were manually inspected by humans to make sure they are correct.

D Per-Target Cost and Time Breakdown

Table 7 reports the per-target cost and time measurements that back the per-project aggregation in Table 4. Each row corresponds to a single Magma target. *Total* is the sum of cost/time over all reports for that target; *Avg. Rpt.* is the average cost/time per analysis report; *Avg. Seed* is the average cost/time per generated seed.

Target	Total		Avg. Rpt.		Avg. Seed	
	Cost ($)	Time (s)	Cost ($)	Time (s)	Cost ($)	Time (s)
PDF011	2.11	584	0.65	89	0.0056	11
PDF018	4.50	622	1.43	118	0.0070	9
PDF021	3.76	692	1.18	105	0.0074	13
PHP004	4.19	912	1.32	139	0.0077	17
PHP009	4.21	889	1.33	122	0.0071	17
PNG001	3.78	643	1.19	99	0.0073	12
PNG007	5.47	730	1.74	117	0.0079	13
SND017	5.90	726	1.92	142	0.0045	10
SND020	3.95	684	1.27	128	0.0047	10
SQL002	6.71	584	2.21	131	0.0031	6
SQL003	7.01	1121	2.29	109	0.0051	26
SQL012	12.01	503	3.98	131	0.0022	4
SQL013	4.60	531	1.48	95	0.0052	8
SQL014	8.69	563	2.86	114	0.0032	7
SQL015	5.59	544	1.83	124	0.0036	6
SQL020	4.05	511	1.31	107	0.0035	6
SSL001	4.67	530	1.50	115	0.0060	6
SSL020	3.78	602	1.20	103	0.0060	10
TIF002	5.55	723	1.77	124	0.0082	12
TIF008	6.07	723	1.95	114	0.0068	13
TIF014	4.76	1022	1.48	155	0.0100	19
XML003	4.53	530	1.46	114	0.0049	6
XML009	5.49	631	1.78	130	0.0055	8
Avg.	**5.28**	**678**	**1.71**	**118**	**0.0058**	**11**

Table 7: Cost and time analysis per target (Cost in USD, Time in seconds).

E Sample Analysis Report for PDF018

This appendix reproduces the analysis report SeedSmith's agent produced for the `PDF018` case study (Section 6.3.1), with reformatting for readability. Boilerplate fields that overlap with the main paper (sink source, harness entry point, full call trace) are omitted; we keep the agent's reasoning steps, prior-round feedback, condition analysis, seed-construction strategy, and the Python script that produced the one-shot crashing seed.

Reasoning for Path Choice

- **Step 1.** Searched `AnnotInk::` to understand the class structure; found the constructor, `parseInkList`, and `draw` methods, revealing that `inkList` is an array of `AnnotPath` pointers that can contain null entries.
- **Step 2.** Searched `parseInkList`; the function body exposed the critical pattern:

```
void AnnotInk::parseInkList(Array *array) {
    inkListLength = array->getLength();
    inkList = (AnnotPath **)gmallocn(inkListLength, sizeof(
      AnnotPath *));
    memset(inkList, 0, inkListLength * sizeof(AnnotPath *));
    for (int i = 0; i < inkListLength; i++) {
        Object obj2 = array->get(i);
        if (obj2.isArray())            // only array entries
      populated
```

```
            inkList[i] = new AnnotPath(obj2.getArray());
        // non-array entries leave inkList[i] = null
    }
}
```

- **Step 3.** Searched `getCoordsLength`; `AnnotPath::getX/getY` have bounds checks, but the issue is dereferencing a null `path` pointer *before* those methods are called.
- **Step 4.** Searched `render_page` and `displayPageSlice` to trace the call path from harness to sink, confirming the standard PDF rendering pipeline.
- **Step 5.** Searched `Page::displaySlice` to locate the exact `annot->draw()` call site.

Conditions Analysis

Critical crash point: `const AnnotPath *path = inkList[i]; if (path->getCoordsLength() != 0)` dereferences `path` with no null check.

Root cause. `parseInkList` (a) allocates `inkList` and zero-initializes via `memset`; (b) populates only entries where `obj2.isArray()` holds; (c) leaves slots for non-array entries (null, numbers, strings, etc.) at zero; (d) `AnnotInk::draw` subsequently iterates and dereferences `inkList[i]` without a null check. The triggering conditions are: a null `inkList[i]` from a non-array `/InkList` element, the direct dereference in `draw`, and an `inkListLength` that includes the null slot.

Seed-Generation Strategy

The crash is triggered by a PDF with an `Ink` annotation whose `/InkList` mixes a valid coordinate array with a non-array entry (`[valid_array, 123]` or `[valid_array, "s"]`, etc.). `parseInkList` creates an `AnnotPath` for the first entry and leaves the second null; `draw` then crashes at `path->getCoordsLength()` on the second iteration.

F LLM Prompts Example

System prompt of Code Exploration Agent is shown in Figure 9.

System Prompt (Code Exploration)

```
You are a security expert specializing in vulnerability research. Your task is to generate a detailed and precise report on
how to trigger crashes in sink functions within a target codebase. Your report will be used by a downstream team with NO
PRIOR KNOWLEDGE OF THE TARGET OR THE CRASH. Therefore, it must include ALL
NECESSARY INFORMATION:
1. Code snippets
2. Step-by-step explanations
3. Analysis of control-flow paths, relevant conditions, and seed generation strategies
4. Clear instructions for reproducing the crash
## Essential Guidelines:
Tool Usage:
+ Use the provided grep tool (grep -rnE <expression>) to thoroughly analyze the entire source code directory.
+ Employ generalized, yet precise patterns. For instance, use patterns like ->param instead of specific references like
  obj->param to ensure broader and more inclusive search coverage.
+ Avoid reusing patterns from previous searches.
## Report Structure:
Your report must be structured to include the following sections:
1. Sink function details: location, signature, purpose
2. Harnesses: file names, line numbers, entry points
3. Call trace: from harness to sink (if provided), or discovered via analysis
4. Relevant conditions: if, switch, etc. on the path
5. Seed generation strategy: Use libraries (base64, zlib, etc.) where applicable, and Python code (or pseudocode) for
   long/complex seeds. The size of the seed MUST be under 2MB.
6. IMPORTANT: THIS GENERATED REPORT MUST NOT CONTAIN REPETITIVE PATTERNS TO DEMONSTRATE A SEED THAT COULD EXPLOIT THE
   VULNERABILITY. IF YOU HAVE TO SHOW SUCH EXAMPLES, PLEASE PROVIDE A SHORTER SUMMARY.

After completing the analysis, you MUST output a report in the specified format. Your final report must be
meticulously structured, explicit, and thorough to enable seamless seed generation and crash reproduction by
the downstream team.
<report>
    <sink_function_details>
        - Location: file path
        - Signature: function declaration
        - Purpose: functionality description
        - Code: complete code snippet of the sink function
    </sink_function_details>
    <harnesses>
        - File Names:
        - [file_name]:[line_number] (entry point)
        - Clearly describe each harness entry point.
        - These are potential harnesses that can be used to trigger the sink function. You can only use one harness per time.
    </harnesses>
    <call_trace>
        - Step-by-step path from harness to sink function
        - Mention explicitly any optimized or omitted intermediate functions
        - MUST Include complete code snippets for each step in the call trace, DO NOT OMIT ANY CODE NECESSARY
        - Use analysis tools as needed to clarify uncertain paths
        - Do not use __connector__ here, use a real path instead
    </call_trace>
    <reasoning_for_path_choice>
    - Justify the selected call trace path over alternatives.
    - Explain step-by-step how you determined this path as the crash path using tool calls:
        - Step 1: patterns to grep, useful information in grep results, why you choose this pattern
        - Step 2: patterns to grep, useful information in grep results, why you choose this pattern
        - ...
    </reasoning_for_path_choice>
    <conditions_analysis>
        - Detailed breakdown of relevant control-flow structures (if, switch, loops, etc.)
        - Mention explicitly any necessary conditions
        - Explain condition reachability and necessary states to trigger crash
    </conditions_analysis>
    <seed_generation_strategy>
        - Short seed example with explanation for expanding to full seed
        - Python code or pseudocode clearly demonstrating complex seed generation
        - Recommend libraries (e.g., base64, zlib) for accurate and simplified seed generation
    </seed_generation_strategy>
    <script_example>
        # Python script clearly illustrating crash seed generation, the crashing input should be writen into
          /work/crash.txt using f.write().
    </script_example>
    <conclusion>
        Summarize key findings, critical conditions, and confirm steps to reproduce crash reliably.
    </conclusion>
</report>
##Information Provided:
This project is {project name}, in {programming language} , you will also receive:
+ Sink function index (<sink_index>), sink function filename (<sink_file_name>), sink function name (<sink_function_name>),
  and sink function code (<sink_function_code>)
+ The harnesses (<harnesses>) code(s) used to reach the sink.
You currently have no information about the call trace. When using the tool, begin your analysis and using the tool by
treating the sink function as the entry point.
```

Figure 9: System Prompt of Code Exploration Agent