

Testing the Verifier: Automated Testing for Isabelle

Enzo Bestetti¹, Bartosz Gowacki¹, Aravindh Kaneshalingam¹, Jiangjing Xu¹,
Mohammad Abdulaziz¹, and Karine Even-Mendoza¹

King’s College London, *UK*⁽¹⁾
`mohammad.abdulaziz, karine.even-mendoza@kcl.ac.uk`

Abstract. We present a fuzzing-based approach for testing Isabelle by targeting components connected to its trusted core. Four students investigated different subsystems over six months, uncovering 24 distinct anomalies. We reported 16 findings to developers, of which 11 were confirmed. We further wrote 10 patches, resulting in 6 accepted pull requests, thus improving the quality of the wider Isabelle ecosystem beyond its core.

1 Introduction

Proof assistants, like Isabelle, are well-established systems used to build reliable software, hardware, and mathematical libraries. Much of the appeal of proof assistants relies on the ‘correctness’ guarantees they provide. The most prominent aspect of proof assistants’ correctness (in terms of literature) concerns their soundness i.e. the guarantee that they may only prove true statements. Indeed, because of the relative criticality of the soundness of a proof assistant, much work has been dedicated towards insuring and analysing their soundness.¹ Another aspect of a theorem prover’s correctness is that of the reliability of a theorem prover and its software ecosystem: for instance, does the proof assistant crash or hang? Do the tools supporting it (e.g. compilers) work reliably? Although reliability is less critical, it is still very relevant given that the unreliability of a proof assistant would substantially hinder its usability.

Fuzzing has proven effective for testing the reliability and robustness of large software systems in practice [18,49]. It has been widely applied across domains, including databases [25], communication protocols [6,37], compilers [20,30,43,47], simulators [18,27], and general-purpose software [21,22]. Compared with many formal methods techniques, fuzzing is often less constrained by issues such as state-space explosion and the need for precise formal specifications, but may face the oracle problem [9].

Motivation We believe applying modern testing methods to a proof assistant like Isabelle could be helpful in a number of ways: 1. it could unveil soundness

¹ Notably for Isabelle, Kuncar’s work on definitions in Isabelle/HOL [29]; Rocq developers keep a list of (unfixed) issues, some of which are soundness findings [3]; interested readers should consult Paulson’s [1] or De Moura’s [4] blog posts.

issues that were not encountered, 2. it could reveal crashes and hangs, which, if left without fixing, would substantially reduce the usability of the system, and 3. it can unravel 'scenarios', which are not necessarily soundness findings, but which were not taken into account while designing and implementing the system and/or its features.

Challenge Although automated testing has seen many applications, it was never systematically applied to proof assistants. This is because proof assistants pose challenges, some of which are uncommon to other types of software systems. For instance, a proof assistant like Isabelle is a software system composed of a large and diverse stack of software: the kernel, the GUI, external solvers, solvers/provers implemented in Isabelle/ML, and the PolyML compiler. Furthermore, similar to many other complex software systems, due to the extensive context (a user, operating system, network connection, etc) in which a proof assistant operates, Isabelle lacks a formal (or otherwise complete) specification of its expected behaviour. Indeed, due to this lack of specification, for many of the issues in the Isabelle ecosystem highlighted as 'findings' by the system's users it is reasonably disputed whether any of them are 'findings' or problematic at all: it is plausible that most of those findings would only affect inexperienced or malicious users, i.e. they are not findings assuming the user has the tacit knowledge.

At the technical level, due to the size of the search space, it would be close to impossible to test a system like Isabelle without dividing it into components, which is the most common testing method, known as blackbox testing. Furthermore, due to the lack of expected behavioural specification of the system overall and its different components, a challenge with testing the system's components separately is that it will not be straightforward to do that in a way which yields information helpful in deriving conclusions about the whole system's behaviour, making creating *harnesses* challenging.

Contributions In this work we report on preliminary work to apply automated testing to investigating the reliability of Isabelle and its software infrastructure. We focus on testing Isabelle's GUI (jEdit), the PolyML compiler, and the sledgehammer encodings to external SMT solvers, with the overall aim of testing systems' reliability (e.g. whether they crash or hang) rather than Isabelle's soundness. Another aim was finding 'interesting scenarios', which would alert Isabelle's users and developers to unforeseen corner cases, thus improving the overall understanding of the system's behaviour.

Open Science Statement. Our code, bug reports, and fuzzing data and scripts are in [13,14,15,26,24,45].

2 Background

Fuzzing, a widely used automated random testing technique where the System Under Test (SUT) is executed with large numbers of generated inputs (commonly known as test inputs or seeds), has been greatly successful in discovering correctness, robustness, and security issues on various SUTs [18,20,25,27,30,32,37,43,46,48,49].

Fuzzing can be categorised by the level of knowledge of the system exposed to the fuzzer. In *blackbox fuzzing*, inputs are generated without any knowledge of the internal structure or execution of the system under test (SUT) [16,33,47]. *Graybox fuzzing* leverages lightweight runtime feedback, such as code coverage or crash information [18,20,22,37] or entropy [11,19], to guide input mutation, seed prioritisation and exploration. *White-box fuzzing* assumes full SUT’s knowledge, applying techniques such as program analysis or symbolic reasoning to systematically guide input generation [12,23,46]. In recent years, AI-powered and LLM-based testing frameworks, including agentic systems such as WhiteFox [46], Fuzz4All [43], ChatAFL [32] and SearchSYS [18], have incorporated generative AI into the fuzzing and testing process to improve test-case diversification, semantic validity, and code coverage, via request to generate seeds directly [43,18] or by enhancing the analysis [46,32]. Some of the fuzzing techniques explored in this work investigate the use of LLMs and agentic systems within the fuzzing process, evaluating their effectiveness in the context of theorem-prover tooling.

One of the most widely used graybox fuzzers is *American Fuzzy Lop* (AFL) [49], and its actively maintained fork AFL++ [22]. AFL++ is a mutation-based fuzzer. It maintains an input corpus (seeds). It iteratively generates new seeds, typically via byte-level mutations, with coverage feedback to identify interesting seeds that exercise undercover and previously unseen program behaviour. This approach has been shown to be highly effective across a wide range of domains [6,17,18,37]. *Instrumentation* enables feedback-driven fuzzing by injecting extra code that records execution behaviour, such as code coverage and runtime events. AFL++ uses this information to guide seed diversification via mutations, seed selection and scheduling. We utilise AFL in this work whenever applying graybox fuzzing.

In practice, fuzzing relies on *test oracles* to determine whether a given execution reveals a potential defect. Most common oracles include crashes, hangs, and other forms of abnormal termination, such as memory safety violations and undefined behaviours, typically already embedded in the fuzzing framework [21,49]. Such behaviours are treated as indicators of incorrect or unstable execution, since well-behaved SUTs are not expected to exhibit them under valid operation. Another important form of test oracle is *differential testing*, which is capable of detecting logical defects beyond crashes and hangs. Differential testing compares the behaviour of a SUT under semantically equivalent conditions, such as across independent implementations, compiler optimisation levels, architectures, or software versions. Behavioural inconsistencies observed across these executions may indicate bugs in the SUT.

3 Methodology

Rather than attempting to fuzz or validate the Isabelle/HOL ecosystem as a monolithic whole, which is impractical due to its size and complexity of logic and interfaces, we identify and isolate key interaction points within the system and test them independently. Each interaction point corresponds to a well-defined

component boundary, such as the Poly/ML compiler, unifier, the graphical frontend, etc. We then clearly and formally define the input and output structure for each interaction point. Afterwards, we create a harness, a set of mutators, and then coverage guidance. This approach allows us to do fuzzing only on the integration point, one component at a time, rather than the system as a whole, which should be more effective and efficient.

To operationalise our approach, Isabelle was partitioned into multiple such interaction points and assigned to different investigators (undergraduate students). Each investigator independently designed and implemented a testing pipeline tailored to their assigned component. While the specific techniques vary, all pipelines follow a common pattern:

- **Test Harness.** Construction of a harness to isolate the component;
- **Test Input Diversification Strategy.** Development of input generation or mutation strategies (grammar-based, random, or coverage-guided);
- **Application of Automated Testing Framework.** Design or adoption of existing automated testing techniques, e.g., fuzzing, differential testing, sanitiser-based execution, binding together the test harness and the diversification technique under a single framework, such as AFL or blackbox grammar generators; and
- **Analysis of Results.** Mostly, manual analysis of discrepancies or crashes.

This distributed methodology enables parallel exploration of the system while maintaining methodological freedom per interaction point. Thus, it allows each component to be tested using techniques best suited to its characteristics (e.g., specification-based oracles for lexers, sanitiser-guided fuzzing for low-level runtime code). We discuss next each of the methodologies developed for the interaction points we targeted.

3.1 Poly/ML’s Lexer under X86

The Poly/ML compiler [2], a long-maintained implementation of Standard ML (SML) since the early 1990s, translates Standard ML programs into executable code and is written entirely in SML. Moreover, the Poly/ML compiler is a core component of the ISABELLE/HOL proof assistant toolchain, used to compile most of Isabelle/HOL’s code. As a compiler targeting multiple architectures, it is particularly relevant to study the widely used x86 architecture, where most development and deployment environments are based.

The system under test is the lexical analyser of Poly/ML. The lexer is implemented as a hand-written recursive descent scanner residing in the `LEX_.ML` functor, covering identifiers, numeric literals (integers, reals, hex, words), string and character constants, comments, and reserved words. Despite its central role in a critical verification toolchain, Poly/ML’s lexer has not previously been subject to systematic testing or coverage-guided analysis. In particular, there is no publicly available coverage baseline or prior fuzzing study of this component.

The *specification-based differential testing of Poly/ML’s lexer* methodology combines three complementary techniques: a specification-derived oracle, grammar-guided corpus generation, and coverage-guided fuzzing.

Oracle. The oracle, VERILEX, is a novel implementation of the Sulzmann & Lu derivative-based lexing algorithm [40], which has been formally verified to produce POSIX matches [40,8]. Two implementations are provided: a *reference implementation* generated directly from Ausaf et al.’s mechanised proof in Isabelle/HOL (via its code generation to Scala) [7], and a *manual implementation* in Kotlin that follows the published algorithm [40,35]. The dual-oracle design mitigates a bootstrapping concern: the reference implementation breaks free of human transcription error but shares a trust root with Isabelle/HOL, whereas the manual implementation is independent of Isabelle/HOL but susceptible to human error. The oracle tokenises SML source according to regular expressions derived from ML97 (the Definition of Standard ML, Revised) [34]. These regular expressions constitute the first publicly available regex-based encoding of the SML Core lexical syntax.

Grammar-guided generation. The initial seed corpus is produced by SMLGEN, a grammar-based generator that encodes the SML Core grammar as composable generator combinators. A depth-bounded probabilistic recursion model (quadratic decay of recursion probability with depth) guarantees termination while preserving structural diversity. Five configurable complexity tiers control the character of generated programs. In our evaluation, we observed that a corpus of 100 seeds suffices to exercise all “happy paths” in Poly/ML’s lexer, corresponding to 62.50% of the manually instrumented branch points.

Coverage-guided fuzzing. The generated seeds are fed to AFL++[22] for coverage-guided mutation. Because Poly/ML compiles SML to native code via its own compiler pipeline (no LLVM-IR or GCC-RTL stage), AFL++’s standard compile-time instrumentation cannot be applied. Instead, *manual source-level instrumentation* was performed: 104 trace points were inserted at every branch in the LEX_ functor, each calling a C shim that writes into AFL++’s shared-memory bitmap directly via Poly/ML’s *foreign-function interface* (FFI). The instrumented lexer is observationally equivalent to the original lexer, which is guaranteed by the semantics of SML regarding evaluation of sequential statements. Trace IDs were verified to be collision-free in AFL++’s bitmap.

Differential comparison. After fuzzing, our novel post-fuzzing tool, DIFFCOMP, feeds each input from AFL++’s queue through both POLYLEX (the instrumented Poly/ML lexer) and VERILEX, compares the resulting token streams, and classifies any discrepancies into five categories (oracle-only tokens, poly-lex-only tokens, token-type mismatch, token-text mismatch, wrong token position) using *longest common subsequence* (LCS) alignment.

Pipeline orchestration. The 4 stages (generation, fuzzing, comparison, coverage analysis) are sequenced by an orchestrator to support multi-campaign execution with deterministic hierarchical seeding (each campaign seed is derived via SHA-256 from a master seed and campaign index), crash-safe resumability, and automated analytics. Stages communicate exclusively through the filesystem.

3.2 Poly/ML Compiler Frontend on ARM64

ARM64 is becoming a dominant platform for developer workstations (Apple Silicon) and cloud infrastructure (AWS Graviton). Poly/ML contains ARM64-specific C++ code in `libpolym1/arm64.cpp` for calling conventions, stack frame management, and a native code generator. Poly/ML forms part of the Isabelle trusted computing base: it compiles the SML source of the Isabelle/HOL proof assistant, meaning a latent defect in Poly/ML can silently undermine the reliability of machine-checked proofs. There has been no fuzzing tool previously targeting Poly/ML on ARM64, leaving these architecture-specific components empirically unexamined.

In this section, the system under test is the Poly/ML compiler frontend, comprising the lexical analyser (`LEX_.ML`) and recursive descent parser (`ParseDec.ML` and related modules), together with the C++ runtime library (`libpolym1`) invoked during frontend execution, tested under ARM64 architecture. We follow a coverage-guided fuzzing methodology, applying AFL++ [22], a community-maintained fork of the original AFL [49], with ARM64-native runtime sanitiser instrumentation and a two-phase seed corpus [26] designed to target the lexer and parser components in sequence.

Direct binary fuzzing. AFL++ is applied directly to the `poly` binary in fork-server mode, which reads an SML test program from standard input. An alternative harness-based approach was evaluated first: a C wrapper that invoked `poly` as a child process via `system()`. AFL++ cannot track coverage inside a child process spawned via `system()`, so this approach yielded only 12 distinct coverage edges. Direct fuzzing of the `poly` binary, compiled with `afl-clang-lto` in LTO mode with LLVM PCGUARD coverage instrumentation, produced 2,014 edges after Phase 1 (the first three-day campaign on Subset A; see Table 1). The shift to direct fuzzing is the principal engineering decision in the pipeline design.

Runtime sanitiser instrumentation. Poly/ML is built without compile-time sanitisers. The bootstrap process, in which Poly/ML compiles itself using an earlier binary, is incompatible with AddressSanitizer’s shadow memory model. Sanitisers are therefore applied at runtime via AFL++ environment variables (`AFL_USE_ASAN=1` and `AFL_USE_UBSAN=1`), instrumenting the binary without modifying the build system. UndefinedBehaviorSanitizer is particularly effective on ARM64: the architecture enforces stricter alignment requirements and calling conventions than x86-64, meaning undefined behaviour that manifests as a silent no-op on x86-64 can produce a fatal diagnostic on ARM64. This asymmetry is the principal motivation for ARM64-native testing rather than relying on existing x86-64 evaluation.

Seed corpus and phased campaign design. A corpus of 72 Standard ML programs was written manually and partitioned into two subsets by syntactic target [26]. Subset A (35 seeds across the `basic/`, `operators/`, `edge-cases/` and `regression/` categories) exercises lexer-level constructs: numeric literals, nested comments, operator precedence, and known UBSan regression triggers. Subset B (37 seeds across the `stress/`, `modules/` and `datatypes/` categories) exercises parser-level constructs: deeply nested structures, functor applications, complex

algebraic datatypes, and mutual recursion. Two sequential three-day campaigns were conducted: Phase 1 on Subset A and Phase 2 on Subset B.

Phase 2 incorporated several targeted amendments informed by Phase 1 results. The per-execution timeout was reduced from 10,000 ms to 5,000 ms to improve throughput without missing genuine hangs. AddressSanitizer leak detection was disabled (`ASAN_OPTIONS=detect_leaks=0`) to remove overhead irrelevant to the crash-finding objective. CMPLOG comparison logging was activated on two of the four fuzzer instances to assist AFL++ in solving structured integer and pattern checks in the parser. A third instance used AFL++’s *rare* power schedule, which focuses mutation effort on infrequently-hit edges to diversify exploration. Phase 2 was further augmented by the evolved Phase 1 corpus and Standard ML files extracted from the Isabelle/HOL source tree, bringing the starting corpus to 838 seeds.

Coverage saturation criterion. Coverage growth is sampled every 30 minutes, computing the number of new AFL++ edges discovered per hour. Saturation is declared when fewer than ten new edges are found per hour for three consecutive hours, providing a data-driven stopping criterion rather than a fixed campaign duration. A separate LLVM source-coverage build, compiled with `-fprofile-instr-generate` and `-fcoverage-mapping`, generates per-file region and line coverage reports from the evolved AFL++ queue after each campaign, quantifying the reachable fraction of `libpolym1/` independently of AFL++’s edge bitmap. We report results in §4.

Grammar-aware mutator retry. Following the observation that Phase 1 and Phase 2 converged on the same coverage ceiling, a 24-hour retry campaign was conducted to test whether a structure-aware mutation strategy could break through it. A custom Python AFL++ mutator (`scripts/sml_mutator.py`) was written to produce SML-specific mutations: pathological floating-point exponents, integer boundary values, deeply nested expressions, long identifiers, and comment corruption. Two of the four fuzzer instances used the grammar-aware mutator; the remaining two retained the default AFL++ strategy as a comparison baseline. The retry reached 2,003 edges and 28.59% region coverage of `libpolym1/`, confirming that the coverage ceiling is independent of mutation strategy and reflects structural unreachability rather than insufficient mutation diversity.

3.3 The Isabelle/HOL Graphical Frontend

In this section, we describe the methodology for testing Isabelle/HOL/jEdit frontend layer. The frontend layer comprises the *Isabelle/Scala connector* and the *Isabelle/jEdit interface*. Despite Isabelle’s role in high-assurance verification (e.g. seL4, the Kepler conjecture proof), no prior systematic testing or public coverage baseline existed for the frontend layer. Isabelle/Scala acts as the mediation layer between the trusted Isabelle/ML kernel and the jEdit text editor via the Prover IDE (PIDE) protocol over authenticated TCP sockets. It is responsible for coordinating document model updates, managing asynchronous proof processing, and rendering results within the interactive development environment.

The framework targets the surrounding layers of Isabelle logic and kernel, where defects manifest as Pollack-inconsistencies [42] in which the displayed state contradicts what the kernel actually computed. The Isabelle/ML kernel and Isabelle/HOL logic are treated as a trusted core and excluded from the SUT. To find issues in the above layers, the framework combines three complementary sub-frameworks, each targeting a distinct class of frontend defect: (1) blackbox differential testing with mutation fuzzing, (2) whitebox testing guided by LLM-assisted source-code scanning, and (3) a three-agent autonomous testing architecture.

Framework 1: Blackbox differential testing with mutation fuzzing. The differential framework detects inconsistencies between the PIDE frontend and the batch-mode oracle by feeding the same mutated theory to both and comparing their token streams. Seeds are drawn from the Archive of Formal Proofs (AFP), which provides syntactically valid theories that exercise non-trivial Isabelle/HOL constructs without requiring synthetic generation.

Mutation operators. Seven mutation operators are applied to the seed corpus, each targeting a distinct lexical or logical surface: conjunction-to-disjunction swaps ($\backslash\text{and}\rangle \rightarrow \backslash\text{or}\rangle$), negation removal, random line deletion, random line duplication, universal-to-existential quantifier weakening ($\backslash\text{forall}\rangle \rightarrow \backslash\text{exists}\rangle$), substitution of `by simp` (and analogous tactics) with `by sorry`, and inequality direction flips. A regex-based scanner locates the `begin` and `end` keywords and restricts mutation to the body, preserving the theory header and trailing `end` so the file is not rejected for trivial structural reasons.

Oracle. The ground truth is established by invoking `isabelle build` on a temporary session whose `ROOT` file is generated automatically from the theory’s `imports` block. Errors are extracted from the oracle’s plain-text log using two regex patterns matching `At command` and `Failed to finish proof` messages, filtered by file path so that errors in imported library theories do not contaminate the verdict.

Server interaction. The frontend path is exercised through the Isabelle server, accessed via authenticated TCP socket. Each worker establishes a session with the `session_start` command, issues `use_theories` for each mutated file, and parses the structured JSON response (handling both top-level error arrays and node-level message lists). Server and oracle execute in parallel via a `ThreadPoolExecutor` so total iteration time is bounded by the slower of the two rather than their sum.

Comparator. The comparator classifies each iteration into one of five outcomes: both pass (match), both fail at the same lines (match), both fail at different lines (mismatch), oracle pass with server fail (server false positive), and oracle fail with server pass (server false negative). Cases where the server reports a strict superset of the oracle’s errors are treated as “more verbose” rather than mismatches, suppressing a known false-positive class.

Framework 2: Whitebox testing with LLM-assisted source scanning. The whitebox framework links directly against Isabelle’s internal Scala libraries via `sbt` and exercises individual components at the API level, targeting con-

currency defects that blackbox methods cannot reliably trigger. Candidate locations are identified by feeding sections of `src/Pure/` and `src/Tools/jEdit/` to a generative language model (Claude Opus 4.6) primed to flag thread-safety hazards, exception-handling defects, and other detectable code vulnerabilities. Every flagged candidate is then verified manually through three checks: reading the surrounding code in context, consulting the file’s `git` history to rule out documented-but-counter-intuitive design choices, and cross-referencing any external library behaviour against its specification. Surviving candidates are transformed into standalone Scala tests, each a deterministic `main` method that exercises the suspected defect and emits a binary verdict. For timing- or interrupt-sensitive findings, the relevant check is wrapped in a 10,000-iteration loop and reports the iteration at which the defect first triggers, providing a more informative artefact than a binary outcome. Once a test reliably fails, a fix is developed in place, the patched build is produced via `isabelle build -b Pure`, and the test is rerun; only a transition from fail to pass (or, for iteration-counted tests, full 10,000-iteration survival) is accepted as evidence of correctness.

Framework 3: Agentic AI testing. The agentic framework instantiates three specialised LLM agents (Mistral Large via API) that cooperate to generate, execute, and evaluate adversarial theory files autonomously, without a fixed ground truth. The *Strategist* maintains a long conversation history and proposes attack strategies (typedef abuse with empty carriers, conflicting type-class instantiations, overloading conflicts, locale tricks, ML code injection, and historical soundness findings); it is explicitly forbidden from instructing the use of `sorry` or `oops`. The *Coder* translates each strategy into a complete, syntactically valid theory file, maintaining its own history to avoid repeating failed attempts. The *Vision* agent is a stateless multimodal observer that receives a single screenshot of the Isabelle/jEdit window and returns a structured description of the visible state (process status, error markers, success indicators), acting as the system’s eyes without the strategic context of the other agents. The GUI interaction layer launches Isabelle/jEdit as a subprocess, locates the window by title, and injects code via clipboard rather than character-by-character typing to avoid latency and encoding artefacts. A cheat-detection substring check inspects each generated theory for `sorry` or `oops` (case insensitive) and forces a retry if found, preventing the system from trivially declaring a contradiction by admitting the goal.

3.4 Sledgehammer’s Interface to External ATPs

In this section, we describe the methodology of testing Sledgehammer with the external ATP interface. Sledgehammer [36] is ISABELLE/HOL proof assistant flagship proof-automation tool. It converts higher-order goals into the first-order TPTP language [41], sends them to external *automated theorem provers* (ATPs), and reconstructs any returned proof into a native Isabelle tactic. The interface links the kernel to multiple external provers through three serialisation layers. The *outbound* TPTP problem written by `atp_problem.ML`, the *inbound* TSTP proof read by `atp_proof.ML`. The SMT-LIB exchange handled by

SMT_Solver.ML. A failure at any boundary reduces Sledgehammer’s success rate without affecting kernel correctness, and is invisible to Isabelle’s own regression suite since they check kernel logic, not external auxiliary parsers.

We test the integration boundaries between Sledgehammer and two complementary ATPs: Zipperposition [10] (approximately 22,000 lines of OCaml, higher-order superposition calculus) and E prover [39] (approximately 145,000 lines of C, first-order saturation). They were chosen for complementary language coverage—a memory-safe ML and an unsafe systems language—and complementary logic. Both are defaults in the Isabelle2025-2 distribution: E is bundled at version 3.2.5-ho (x86_64 via Rosetta on ARM macOS) and Zipperposition at version 2.1. To our knowledge, there is no published coverage or fuzzing baseline of the Sledgehammer or external ATP interface. In our evaluation, we only report findings reachable through normal, valid Sledgehammer inputs, while reporting malformed inputs as additional observations, as these do not affect a user driving Sledgehammer through its supported.

The methodology applies five strategies in sequence, each motivated by limitations of the previous. We discuss each in detail below.

Source-informed targeted fuzzing. Manual audit of `atp_problem.ML` and the Zipperposition TPTP lexer (`Lex_tptp.mll`) identified locations where parser or serialiser assumptions appeared inconsistent (e.g., keyword-handling functions defined for the DFG branch, yet stubbed to the identity for the TPTP branch (line 992 of `atp_problem.ML`). Targeted Isabelle theories were then constructed to drive Sledgehammer through the suspect code paths.

Differential testing across provers. The same Sledgehammer-exported TPTP problems were dispatched to E, Vampire, and Zipperposition in parallel, and discrepancies in which prover crashed, rejected the input, or returned a result were used as oracles for prover-specific defects. Vampire [28]² served as a parsing reference because of its mature TPTP implementation.

AFL++ coverage-guided fuzzing. Coverage-guided fuzzing of an AFL-instrumented Zipperposition build using Sledgehammer-generated TPTP seeds to detect parser crashes, hangs, and robustness issues.

SMT differential fuzzing. Grammar-aware SMT-LIB generation comparing CVC5 versions to detect soundness divergences and solver inconsistencies.

Proof reconstruction testing. Automated proof reconstruction testing across the Sledgehammer pipeline to detect reconstruction failures, prover-wrapper inconsistencies, and defects in ATP-to-Isabelle proof translation.

4 Evaluation

We evaluate our approach in terms of *findings*, referring to distinct anomalous behaviours or inconsistencies observed when exercising interactions between the two system components. We discuss below the results of each system component

² Approximately 170,000 lines of C++

in detail: Poly/ML’s frontend under x86 (§4.1) and ARM (§4.2), the Isabelle/-jEdit layer (§4.3), and the interface of Sledgehammer (§4.4). In addition, some results include code coverage analysis, used as a complementary metric to assess the extent of explored execution paths. The following research questions guide our evaluation:

RQ1: Can fuzzing reveal interaction-level inconsistencies between Isabelle’s core and its external tool ecosystem without relying on a formal specification?

RQ2: Which fuzzing strategy most effectively exposes such inconsistencies, in terms of both number of findings and externally confirmed fixes (patches/issue resolutions)?

4.1 Poly/ML’s Lexer Under X86 (Findings #1-#5)

Experimental Setup. Under the x86 architecture, we executed 100 campaigns over 20 hours. Each campaign generated 100 initial SML seeds, which were then fed to AFL++ [22] (commit `afbc07`). We fuzzed for 10 minutes per campaign, after which coverage plateaued at approximately 75% of instrumented branches. We ran our experiments on an x86-64 machine with AMD Ryzen 7, 32 GB RAM, running Arch Linux (rolling release). All experiments were performed under identical conditions.

Results. In total, we analysed 39,733 inputs (from the post-AFL queue), with an overall mismatch rate of 55.50%. No AFL-level crashes or hangs were observed. The high mismatch rate reflects AFL++’s queue-retention policy, which is biased towards divergence-triggering inputs. We discuss below 4 distinct mismatch findings in detail; the last finding (finding #5) can be found in the Appendix of this paper.

Finding #1: Qualified identifiers. ML97 defines a qualified identifier as

$$\text{strid}_1.\text{strid}_2.\dots.\text{id},$$

where each `strid` must be an *alphanumeric identifier*. Poly/ML’s lexer does not validate the `strid` components and greedily extends any identifier through dots. For instance, on input `Foo.+bar`, the lexer produces a single `Identifier` token for the entire chain rather than rejecting the symbolic component `’+’` as invalid:

```
1 [Input]
2 Foo.+bar
3
4 [Expected]
5 Lexical Error
6
7 [Observed]
8 Allows the invalid identifier anyway.
```

Here, validation is instead deferred to the semantic analyser, which only checks whether named structures are declared, but not whether they are syntactically valid structure identifiers. This implies that Poly/ML accepts qualified identifiers that are disallowed by ML97. This discrepancy is likely due to the earlier ML90 definition not including the relevant rule, and the revised ML97 specification (Appendix G, list of all changes) omitting this update. We reported this issue

via private communication with the Poly/ML maintainer (previously unknown, confirmed & patch under resolution by the maintainer).

Finding #2: Reals with fractions. When the input contains a digit followed by three dots (e.g. `9...2`), the expected tokenisation under maximal munch is `Integer(9)`, `Ellipsis`, `Integer(2)`. Poly/ML detects the pattern `9..` where a non-numeric character follows the first dot, declares a malformed real, and halts. The lexer commits to a real-number path and does not backtrack, despite having a backtracking mechanism for analogous cases (e.g., nearly hex word constants like `0wxZffa`, which are correctly tokenised as `Identifier(0wx)`, `Identifier(Zffa)` instead of declaring “malformed hex constant”):

```
1 [Input]
2 val x = 9...2
3
4 [Expected]
5 VAL Identifier(x) = Integer(9) Ellipsis Integer(2)
6
7 [Observed]
8 ERROR (malformed real constant)
```

We reported it to the developers (Issue #256). A pull request fixing this issue was accepted into Poly/ML’s public repository on GitHub (commit `fc87489`).

Finding #3: Character constants. A character constant in SML `#"c"` must contain exactly one *logical character*. Poly/ML’s lexer reuses the string-lexing routine for character constants and does not enforce the single-character constraint. Inputs such as `#"abc"` pass through the lexer unchallenged, propagate through the parser, and only trigger an error during semantic analysis (`Conversion exception (Not exactly one character)`):

```
1 [Input]
2 #"abc"
3
4 [Expected]
5 Lexer error (not exactly one character)
6
7 [Observed]
8 CHAR(abc) token emitted; error deferred to semantic analysis
```

We reported it to the developers (Issue #258). A pull request fixing this issue was accepted into Poly/ML’s public repository on GitHub (commit `2a1c2ab`).

Finding #4: Token class conflation (EtyVar). ML97 distinguishes *type variables* (`TyVar`, one leading prime) from *equality type variables* (`EtyVar`, two or more leading primes). Poly/ML’s lexer omits `EtyVar` entirely and emits `TyVar` for any prime-prefixed identifier. This is unlikely to affect program semantics, as equality is resolved during later compilation phases. We reported these findings via private communication with the maintainer [5], leading to WON’T FIX since the discrepancy does not affect type soundness under later compilation phases.

4.2 Poly/ML Frontend Under ARM64 (Findings #6-#8)

We executed three fuzzing campaigns, summarised in Table 1 under the ARM64 architecture. Four parallel AFL++ instances ran throughout; `fuzzer01` ran as the main instance (-M), `fuzzer02`–`fuzzer04` as secondaries (-S), with periodic corpus synchronisation from `fuzzer01`.

Table 1. Campaign parameters and outcomes. Coverage is region coverage of libpolym1/ measured via a separate LLVM source-coverage build.

Campaign	Dur.	Seeds	Edges	Sat.	Coverage	Crashes
Phase 1 (Subset A)	72 h	35	2,014	~8 h	28.57%	3
Phase 2 (Subset B + evolved)	72 h	838	2,017	~4 h	28.55%	3
Grammar-aware retry	24 h	838	2,003	~8 h	28.59%	0

Experimental Setup. We used commit `dfb7e03` of Poly/ML’s main branch, (the most recent stable release (v5.9.2)). Experiments run on AWS Graviton3 (arm64), using an AWS EC2 `c7g.xlarge` instance (4 vCPUs, 8 GiB RAM, 2 GiB per vCPU) running Ubuntu 22.04.

Runtime sanitiser instrumentation: Poly/ML is built without compile-time sanitisers. The bootstrap process, in which Poly/ML compiles itself using an earlier binary, is incompatible with AddressSanitizer’s shadow memory model. Sanitisers are therefore applied at runtime via AFL++ environment variables (`AFL_USE_ASAN=1` and `AFL_USE_UBSAN=1`), instrumenting the binary without modifying the build system. UndefinedBehaviorSanitizer is particularly effective on ARM64: the architecture enforces stricter alignment requirements and calling conventions than x86-64, meaning undefined behaviour that is silent on x86-64 can produce a fatal diagnostic on ARM64.

Coverage saturation criterion: We sampled coverage growth every 30 minutes, computing the number of new AFL++ edges discovered per hour. We built LLVM with code coverage instrumentation with `-fprofile-instr-generate` and `-fcovgen-mapping`, generating file-level block and line coverage of the fuzzed AFL++ queue.

Results. Phase 1 peaked at 1,178 executions per second before OOM-inducing inputs were added to the corpus. Phase 2 peaked at approximately 190 executions per second due to CMPLOG comparison logging overhead and a larger 838-seed initial corpus. Findings #6 and #8 are discussed here, with Finding #7 covered in the Appendix. No LLM assistance was used.

Finding #6: ARM64-specific unsigned integer overflow. Two valid Standard ML programs consistently trigger an UndefinedBehaviorSanitizer report in the ARM64 code generator. The fault is a signed/unsigned mismatch: dividing by `sizeof(PolyWord)` (type `size_t`) promotes the result to an unsigned type, so on negative offsets, the pointer arithmetic wraps (an undefined behaviour in C):

```

1 [Input]
2 fun fact 0 = 1 | fact n = n * fact (n - 1)
3 val _ = fact 6
4
5 [Expected]
6 Poly/ML compiles and evaluates the function
7
8 [Observed]
9 SIGABRT - UBSan: runtime error: unsigned offset overflow
10 (libpolym1/arm64.cpp:246, CopyStackFrame)

```

This was found manually during pre-campaign seed validation via whitebox, pre-campaign. The defect is ARM64-specific; no equivalent code path exists on x86-64. A local fix casting the divisor to `POLYSIGNED` was validated on Apple M2

ARM64 but has not been submitted upstream. We reported Issue #271 to the developers (previously unknown, unconfirmed).

Finding #8: Module elaboration type safety defect. Phase 2 produced three crash inputs across separate fuzzer instances, producing a segmentation fault in Poly/ML’s module elaboration code. The root cause is a type safety defect in overloading resolution: `allowGeneralisation` in `TYPE_TREE.ML` leaves the overloaded type set uncommitted, allowing the type checker to re-resolve the overload at the call site and invoke integer machine code with floating-point arguments. The crash reproduces without sanitiser instrumentation, confirming a genuine memory access violation. Minimisation identified two distinct reproducers; input was reduced to `fun ma x y = (x - y); ma 0.0 0.0`:

```

1 [Input 1]
2 fun ma x y = (x - y); ma 0.0 0.0
3
4 [Input 2]
5 structure Mat0 = struct val 0 = 0
6 fun s0quare x = x+x fun e 0 = () end;
7 val a = Mat0.s0quare 0.0
8
9 [Expected]
10 Type error (int/real type mismatch)
11
12 [Observed]
13 SIGSEGV (signal 11) on commit dfb7e03b

```

This issue was exposed via fully automated Phase 2 greybox fuzzing. We reported two separate issues to the developers:

(Input 1) Issue #272 (previously unknown, confirmed & fixed): Our GitHub Issue report was confirmed and fixed by the developers (commit `cf7b84a`: *Freeze the type of an overloaded function or literal if it has to be assigned a default type*).
(Input 2) Issue #274 (previously known, closed-as-duplicate): This issue was subsequently closed as a duplicate by the developers, who clarified that the fault is not caused by `val 0 = 0` (a valid SML pattern) but by the overloaded call `Mat0.s0quare 0.0`, which shares the same root cause as Input 1.

4.3 Isabelle/jEdit Frontend Layer (Findings #9-#19)

Evaluation used three approaches: 1) a *whitebox method with LLM-assisted* candidate identification followed by manual verification, 2) a *differential method* comparing PIDE server behaviour against a batch oracle over 200 mutated AFP seeds to detect behavioural inconsistencies, and 3) an *agentic method* using adversarial proof attempts to test system robustness.

Overview of the results: 1) The *whitebox* framework, driven by LLM-assisted candidate identification followed by manual verification, contributed 2 confirmed concurrency issues in Isabelle/Scala and a small set of robustness deficiencies in the XML serialisation layer; 2) the *differential* framework, executed across roughly 200 mutated AFP seeds, surfaced 6 classes of behavioural divergence between the PIDE server and the batch oracle; these are reflect Pollack-inconsistencies in how the IDE communicates that verdict to the user; and lastly, 3) the *agentic* framework produced no new findings against the SUT and confirmed Isabelle’s robustness against adversarial proof attempts.

We discuss the experimental setup and the evaluation in detail below, including the results for the first two campaigns.

Experimental Setup. We target Isabelle2025 on Ubuntu 22.04 with bundled JDK 21 on x86_64 with Intel Core i5-9300H (2.40 GHz) and 32 GB RAM. The first two approaches were executed as separate campaigns on a shared Ubuntu 22.04. The agentic framework ran on a separate Windows host to support screen capture and GUI-based interactions. Each of which utilised 16 GB RAM. LLM-assisted components used the Mistral-large-latest model (March/April 2026) via Mistral API key, including all agentic experiments.

Part-1: Results with Whitebox Testing Campaign (Findings #9-#12).

10+ LLM-driven scans of `src/Pure/` and `src/Tools/jEdit/`, with at least five candidate flags per iteration, were executed, generating at least five candidate issues per iteration. Most AI-suggested candidates were rejected during manual verification; only those passing all three verification stages proceeded to test construction. This process led to two confirmed concurrency issues (in `progress.scala` and `delay.scala`) and two robustness issues (in `xml.scala`), with patches submitted to the Isabelle development team. We discuss one of these findings below, with the rest can be found in the Appendix of this paper (findings #10-#12).

Finding #9: Exception rethrow in `Delay.run()` terminates the global event timer. The catch block in `Delay.run()` (`src/Pure/Concurrent/delay.scala`, line 29) logs non-interrupt exceptions and rethrows them. `Delay` schedules work via a global `Event_Timer` backed by a single `java.util.Timer` singleton. However, the `JDK TimerThread.mainLoop()` has no catch of `RuntimeException` from tasks, thus any escaping exception terminates the timer thread permanently. After this point, all `Delay.invoke()` calls in Isabelle throw `IllegalStateException` or fail silently:

```

1 [Caused By]
2 A Delay whose event throws a RuntimeException, followed by a second Delay.
   invoke() on any other Delay
3
4 [Expected]
5 Logged exception; subsequent invocations succeed.
6
7 [Observed]
8 Timer thread dies; subsequent Delay.invoke() throws IllegalStateException.

```

This impacts syntax highlighting, session readiness notifications, file loading, and other editor operations that rely on the same debounced timer. In practice, the editor degrades silently over time as a session runs. Initial guidance for this investigation came from Fabian Huch (Isabelle developer), who had observed editor slowdowns after extended use, which can be one of the reasons. A fix removing the `throw exn` statement was developed. The patch and accompanying test input were submitted to the Isabelle development mailing list and acknowledged by the maintainers, who confirmed these findings and noted that further time and deeper analysis are required to accept the patch.

Part-2: Results with Differential Testing Campaign: PIDE vs. Batch Mode Divergences (Findings #13-#19).

About 200 runs in separate sessions, each starting from a fresh AFP seed, applying 10 random mutations, then selecting a new seed and sending results to both the server and batch oracle in parallel. Each iteration is bounded by a 1200-second `use_theories` time-

out and a 900-second oracle timeout. Configurable parallel workers operate on disjoint theory namespaces (`Test_0.thy`, `Test_1.thy`, ...) to satisfy isolation requirements. A shared lock-protected restart system reconnects after 5 failed connections, avoiding redundant restarts. About 30 mismatches were flagged; after removing false positives from JSON parsing limits, 6 real frontend discrepancy types remained. In each case, the kernel verdict (pass or fail) is consistent across the two execution paths (PIDE server vs the batch oracle). We discuss one of these findings below, with the rest can be found in the Appendix of this paper (findings #14-#19).

Finding #13: Stale session state across edits. Across six successive mutations of the `Risk_Free_Lending` theory, the PIDE server reported an error at the same line throughout, while the batch oracle reported errors at three different locations spread across the file. The first mutation produced a genuine failure that the live session cached; subsequent `use_theories` invocations on the modified file did not fully invalidate the cached failure. *User impact:* A developer editing in jEdit may continue to see the same red marker on a line after editing a different region of the file, masking the real failure location.

Part-3: Results with Agentic Testing Campaign (No Findings). About 100 runs of the Strategist-Coder-Vision loop, each ending on contradiction, strategy exhaustion, or manual stop. No soundness issues were found; results reflect framework behaviour, not problems in Isabelle’s core system.

The cheat-detection layer correctly intercepted attempts by the Coder to admit goals via `sorry` or `oops`, and ISABELLE/HOL proof assistant correctly rejected the resulting adversarial theories in the dominant majority of cases. The principal limitation observed was the Vision agent: ambiguous `READY` classifications when the prover was still processing, and occasional “description drift” in which the agent reported error messages primed by conversational context rather than present on screen. The results suggest LLM agents work better as input generators than as oracles, and that combining PIDE polling with screenshot analysis would be more reliable than screenshots alone used here.

4.4 Sledgehammer (Findings #20-#24)

We tested Sledgehammer’s interface to external ATPs via several testing and fuzzing techniques. We discuss results per applied method; we used no LLM assistance in this section.

Experimental setup. Experiments ran on a MacBook Pro (14-inch, 2021) with Apple M1 Pro, 16 GiB RAM, macOS Tahoe 26.4.1, against Isabelle2025-2 with its bundled provers. Patches were validated against forked builds of Zipperposition 2.1 and E 3.2.5-ho. The *fuzzing* framework was built on AFL++ 4.35c [22], using OCaml AFL instrumentation for Zipperposition via the `4.14.1+af1` opam switch in an Ubuntu 22.04 Docker environment on macOS arm64. Zipperposition was invoked with a 3-second internal timeout (`--timeout 3`) and AFL++ with a 5-second per-execution timeout (`-t 5000+`), and seeded with 30 Sledgehammer-generated TPTP problems collected during the targeted phases. *Coverage* was

instrumented at two levels. 1) *Zipperposition (OCaml)*: compile-time instrumentation using `bisect_ppx`, with coverage reports generated via `bisect_ppx-report`. 2) *E prover (C)*: compiled with `-fprofile-arcs -ftest-coverage -O0` using Apple Clang on macOS arm64. Coverage was collected via `gcov` (LLVM 21.0.0, compatible with GCC `gcov` 4.2.0 interface) and processed using `lcov` 2.4-0.

Part-1: Results with source-informed targeted fuzzing (Findings #20-#21). This strategy, augmented by replaying selected TPTP-library benchmark problems against an audited unification path, produced two findings. We discuss finding #20 here, and finding #21 in the Appendix of this paper.

Finding #20: TPTP keyword collision. The TPTP serialiser in `atp_problem.ML` (line 992) defines `avoid_clash` as the identity function in the TPTP branch, unlike `avoid_clash_with_dfg_keywords` in the DFG branch. As a result, Isabelle theory constants matching TPTP reserved keywords (`fof`, `tff`, `cnf`, `thf`, `include`) are emitted unrenamed. Zipperposition’s lexer, via the `atomic_word` rule, always classifies these as reserved tokens and rejects them, while E and Vampire accept them due to context-sensitive lexing.

```

1 [Input]
2 theory T imports Main begin
3   consts cnf :: "nat => nat"
4   lemma "cnf 0 = cnf 0" by simp
5 end
6
7 [Expected]
8 Sledgehammer dispatches a parseable TPTP problem.
9
10 [Observed]
11 Zipperposition: "syntax error", proof attempt failed.
```

We reported these findings to the developers of Zipperposition as issue #102 along with a pull request #103, adding the five keyword tokens as alternatives in the `atomic_word` rule. This was acknowledged by the developers, and the PR was merged (a0dc267) (confirmed, fixed, closed). On the Sledgehammer side, an `avoid_clash_with_tptp_keywords` function mirroring the DFG handling was reported to the `cl-isabelle-users` mailing list in the related thread along with a patch [45]; no response had been received at the time of writing.

Part-2: Results with differential testing across provers (Finding #22-#23). Applied to 30 Sledgehammer-exported files, the technique surfaced the third finding we discuss below.

Finding #22: TSTPSkipSource crash on list-form sources . During differential testing on 30 Sledgehammer-exported TPTP files, the E prover binary terminated with exit code 3 on 15 of the 30 files; Zipperposition handled all 30. The failure occurs in `ccl_clauses.c`, function `TSTPSkipSource`, when a TPTP annotated formula carries the optional `source` and `useful_info` fields with a list-form source (`[]`) at the opening square bracket. Sledgehammer routinely emits such annotations to attach Isabelle-specific metadata—for example `[isabelle_non_rec_def, isabelle_rank(900)]`—and the same defect reproduces on both E 3.2.5-ho (the version bundled with Isabelle2025-2) and the development build E 3.3.1-ho.

```

1 [Input]   thf(a_type, type, a: $i).
2           thf(b_type, type, b: $i).
3           thf(test_axiom, axiom,
4             (a = b), [], [some_info]).
5           thf(test_conj, conjecture, (a = b)).
6
7 [Command] eprover-ho --tstp-in --tstp-out
8           --auto-schedule --cpu-limit=10 test.p
9
10 [Expected] E parses the optional source / useful_info fields
11            and proceeds (Zipperposition does so on this input).
12
13 [Observed] test.p:4:(Column 14):(just read '['):
14            Identifier ... expected, but Opening square brace
15            ('[') read. Exit code 3.

```

In our investigation, we found that gcc/lcov coverage showed `ccl_clauses.c` at 0.4% line coverage on the Sledgehammer-driven path versus 33% under E’s default test suite, accounting for the defect’s longevity in shipping releases. We reported these findings to the maintainers of E and Sledgehammer [44] along with a suggested fix and 15 new test inputs, which were accepted as commit `ebb74302` on the E prover `master` branch.

Additional findings, discovered during differential testing over TPTP inputs across multiple theorem provers, are discussed in the Appendix (Finding #23)

Part-3: Results with AFL++ coverage-guided fuzzing (No Findings).

AFL++ produced no further defects. The campaign of AFL++ on the OCaml-instrumented Zipperposition build ran for 70 hours of wall-clock time across 22.9 million total executions (approximately 91 exec/sec, map density 0.17%, stability 46–65%), producing no crashes and 100 saved hangs (all legitimate `ResourceOut` timeouts on lambda-heavy benchmarks, not defects). The negative result is itself a finding: it shows that blind coverage-guided fuzzing is largely ineffective on external reasoning components to the core theorem-proving systems, where deep state exploration requires inputs satisfying non-trivial well-formedness invariants not preserved by random mutation. Considering Parts 1–3, these campaigns support the conclusion that the source-informed strategy is methodologically preferable, rather than a fallback driven by greybox fuzzing byte-level limitations.

Part-4: Results with SMT differential fuzzing (No Findings & Finding #24). A grammar-aware SMT-LIB generator produced approximately 25,000 satisfiability rounds across CVC5 1.2.0 (Isabelle-bundled, native arm64) and a self-built CVC5 1.3.4-dev, checking for soundness divergences between the two versions. The campaign produced no novel defects and no crashes.

Z3 version-lag observation (Finding #24): During SMT differential testing, we discovered an infrastructure-level empirical anomaly. Three SIGSEGV crashes in Isabelle-bundled Z3 4.4.0 were reproducible during SMT differential testing; all three are absent in the current Z3 4.16.0 (Issues #588, #863, #864 and #1445). These are not novel defects—the upstream fixes long predate this work—but the fact that Isabelle ships with a Z3 binary nearly a decade out of date is a noteworthy infrastructure observation that may merit attention from the Isabelle release team. Nonetheless, this suggests that bugs discovered through fuzzing may be triggered by genuine real-world systems such as ISABELLE/HOL proof assistant.

Part-5: Results with proof reconstruction testing (No Findings). A

separate harness, `fuzz_reconstruction.py`, drives ISABELLE/HOL proof assistant through the full Sledgehammer pipeline and traces proof reconstruction attempts, capturing 1,500 planned invocations from the *Archive of Formal Proofs* (AFP).

After parser-side filtering, 666 invocations reached an external ATP, and 260 produced reconstructable proofs (Zipperposition: 126, E: 134). Four false-positive failures were localised to Zipperposition, while 402 invocations did not reach a prover due to wrapper-level invocation failures across Vampire, SPASS, and CVC5. No genuine reconstruction failures were observed, providing a corroborating negative result that suggests the defect search space is concentrated in encoding and external-prover stages upstream of reconstruction.

4.5 Discussion

RQ1. Over six months, four undergraduate students (not Isabelle experts) found 24 distinct anomalies across components connected to the trusted core. The approach relied on methods such as fuzzing, differential testing, and runtime oracles. Of these, 16 findings were reported (11 confirmed). We further wrote 10 patches, resulting in 6 accepted pull requests, that improved the ecosystem.

RQ1 Answer. Yes. The study shows fuzzing to be effective in discovering anomalies even if no formal specification is available. This supports that even relatively low-expertise users can effectively use fuzzing to detect real integration-level bugs in Isabelle’s surrounding toolchain.

RQ2. The largest number of findings came from the Isabelle/jEdit differential testing campaign comparing PIDE and batch mode behaviour. Yet, these findings led to neither fixes nor reports, while whitebox testing of jEdit produced one confirmed report. This may indicate lower community priority or a need to better understand what developers consider important to report.

The highest number of confirmed fixes came from the Sledgehammer interface experiments, with differential testing slightly outperforming source-informed targeted fuzzing. This suggests that Sledgehammer is a critical component interaction within the Isabelle ecosystem. For Poly/ML, the x86 backend attracted more developer attention than ARM64, with four confirmed findings and two fixes already merged, compared to fewer confirmations on ARM64.

Top performers were feedback-related approaches (differential testing, greybox/whitebox). In contrast, grammar-based testing, pure AFL++ fuzzing, and proof reconstruction produced no significant results. The first two likely suffered from low-validity inputs, while proof reconstruction lacked sufficient diversification power to expose inconsistencies.

RQ2 Answer. Effective fuzzing requires targeted feedback mechanisms, and components closer to core logical, lexing and parsing tend to yield more meaningful results and fixes.

5 Conclusion, Vision, and Outlook

The effectiveness of our testing pipeline evolved through the iterative refinement of mutation strategies. While initial campaigns relied on standard AFL++ byte-level mutations, these frequently failed to bypass the initial parsing stages of the Isabelle ecosystem. To address this, we developed domain-specific mutators: for Poly/ML, we implemented grammar-aware mutators that respect SML syntax; for the PIDE frontend, we designed logical mutators (e.g., quantifier swapping) that preserve theory structure while altering proof semantics.

Developing effective harnesses for Isabelle’s diverse stack required overcoming several technical hurdles. We found that direct binary instrumentation of the Poly/ML runtime via a C shim was more effective than high-level wrappers, as it preserved the fine-grained coverage feedback necessary for graybox fuzzing. For the graphical frontend, bypassing the UI layer in favor of direct PIDE protocol interaction via TCP sockets allowed for higher throughput and more deterministic state management. In the future we plan to isolate the serialisation boundaries (e.g. TPTP or SMT-LIB exports) as these provide stable, well-defined entry points for testing tools.

Our methodology is subject to several inherent limitations. The “oracle problem” remains a central challenge; while differential testing provides a partial solution, it cannot verify the ultimate correctness of a prover’s logic without a secondary, independent ground truth. Furthermore, the sheer size of Isabelle’s state space means that even coverage-guided methods only exercise a fraction of the possible execution paths.

In the longer term we would like to develop an automated testing setup/harness for Isabelle. Such a harness could assist in the process of Isabelle’s development, by simply being a part of regression testing. More interestingly, we believe that applying these testing methods to Isabelle will help devise more complete (informal) specifications and expectations of the system’s behaviour. In addition to the potential of finding soundness issues, testing might help find scenarios in which the system could be abused, for instance, to ‘fake’ a theorem.

From the perspective of the theory and practice of testing, we are also interested in developing testing methods specialised for proof assistants, given their unique complexity, as software systems, and the fact that they lack behaviour specification due to the aforementioned reasons. The most immediate direction is creating a library of harnesses and mutators for long-term reuse, improved automation and test oracles (e.g. automation of proof reconstruction in a fuzzing context), and extending cross-component differential fuzzing across ATPs and PIDE, which appears particularly important. Another direction is automating analysis of findings using Gen-AI-aided methods, and C-reduce-style [38] minimisation for structured inputs, including automated bug triage and deduplication. Further work should investigate jEdit more deeply, potentially using ideas from search-based testing [31]. Another longer-term and more foundational direction is developing (semi-)automated methods to split a system like Isabelle into smaller components, along with their interface specifications.

References

1. Broken proofs and broken provers, https://lawrencecpaulson.github.io/2026/01/15/Broken_proofs.html
2. PolyML FAQ, <https://www.polymml.org/FAQ.html>
3. Rocq/dev/doc/critical-bugs.md at master · rocq-prover/rocq, <https://github.com/rocq-prover/rocq/blob/master/dev/doc/critical-bugs.md>
4. Who Watches the Provers? — Leonardo de Moura, <https://leodemoura.github.io/blog/2026-3-16-who-watches-the-provers/>
5. Private communication regarding poly/ml lexer behaviour (2026), email correspondence between David Matthews and Enzo Bestetti
6. Andronidis, A., Cadar, C.: Snapfuzz: high-throughput fuzzing of network applications. In: ISSTA 2022. p. 340–351. ISSTA 2022, ACM (2022). <https://doi.org/10.1145/3533767.3534376>
7. Ausaf, F., Dyckhoff, R., Urban, C.: Posix lexing with derivatives of regular expressions. Archive of Formal Proofs (2016), <https://isa-afp.org/entries/Posix-Lexing.html>, Formal proof development
8. Ausaf, F., Dyckhoff, R., Urban, C.: POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl). In: Interactive Theorem Proving, vol. 9807, pp. 69–86. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_5
9. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering **41**(5), 507–525 (2015). <https://doi.org/10.1109/TSE.2014.2372785>
10. Bentkamp, A., Blanchette, J.C., Cruanes, S., Waldmann, U.: Superposition for lambda-free higher-order logic. In: IJCAR 2018. LNCS, vol. 10900, pp. 28–46. Springer (2018). https://doi.org/10.1007/978-3-319-94205-6_3
11. Böhme, M., Manès, V.J.M., Cha, S.K.: Boosting fuzzer efficiency: an information theoretic perspective. In: ESEC/FSE 2020. p. 678–689. ESEC/FSE 2020, ACM (2020). <https://doi.org/10.1145/3368089.3409748>
12. Cadar, C., Dunbar, D., Engler, D.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI’08. p. 209–224. OSDI’08, USENIX Assoc. (2008)
13. enzoibest: enzoibest/polyfuzz: Polyfuzz testing infrastructure (2026). <https://doi.org/10.5281/zenodo.19593519>, <https://doi.org/10.5281/zenodo.19593519>
14. enzoibest: enzoibest/smlgen: Smlgen - grammar-guided generator for standard ml (2026). <https://doi.org/10.5281/zenodo.19593553>
15. enzoibest: enzoibest/verilex: Verilex - sulzmann & lu lexer for standard ml (2026). <https://doi.org/10.5281/zenodo.19593541>
16. Even-Mendoza, K., Cadar, C., Donaldson, A.F.: CsmithEdge: more effective compiler testing by handling undefined behaviour less conservatively. Empirical Software Engineering **27**(6), 129 (2022). <https://doi.org/10.1007/s10664-022-10146-1>
17. Even-Mendoza, K., Dakhama, A., Berger, H.: Apkfuzz: Search-based fuzzing for android apk vulnerability. LNCS p. 6 (2026), sSBSE 2026 - Challenge Track (Accepted)
18. Even-Mendoza, K., Menéndez, H.D., Langdon, W., Dakhama, A., Petke, J., Bruce, B.R.: Search+LLM-based testing for ARM simulators. In: ICSE-SEIP. pp. 469–480 (2025). <https://doi.org/10.1109/ICSE-SEIP66354.2025.00047>

19. Even-Mendoza, K., Obiri, J., Dakhama, A., McMinn, P., Langdon, W.B.: Fuzz3: Entropy as a third oracle. In: SSBSE 2026 - Challenge Track. LNCS, Springer (2026), https://solar.cs.ucl.ac.uk/pdf/Even-Mendoza_2026_SSBSE.pdf, forthcoming
20. Even-Mendoza, K., Sharma, A., Donaldson, A.F., Cadar, C.: GrayC: Greybox fuzzing of compilers and analysers for c. p. 1219–1231. *ISSTA 2023*, ACM (2023). <https://doi.org/10.1145/3597926.3598130>
21. Fioraldi, A., Maier, D.C., Zhang, D., Balzarotti, D.: Libafl: A framework to build modular and reusable fuzzers. In: *CCS '22*. p. 1051–1065. *CCS '22*, ACM (2022). <https://doi.org/10.1145/3548606.3560602>
22. Fioraldi, A., et al.: AFL++ : Combining incremental steps of fuzzing research. In: *WOOT USENIX Workshop 2020*
23. Ganesh, V., Leek, T., Rinard, M.: Taint-based directed whitebox fuzzing. In: *ICSE '09*. pp. 474–484. *ICSE '09*, IEEE (2009). <https://doi.org/10.1109/ICSE.2009.5070546>
24. Głowacki, B.: Isabelle dissertation artifacts: Differential testing bug reports. GitHub repository (2026), https://github.com/BGlowacki-kcl/Isabelle_dissertation_Bartosz/tree/main/differential/bug_reports, differential testing results (interesting cases)
25. Jiang, Z.M., Bai, J.J., Su, Z.: DynSQL: Stateful fuzzing for database management systems with complex and valid SQL query generation. In: *USENIX Security*. pp. 4949–4965. *SEC '23*, USENIX Assoc. (2023). <https://doi.org/10.5555/3620237.3620514>
26. Kaneshalingam, A.: Coverage-Guided Fuzzing of Poly/ML on ARM64 (2026). <https://doi.org/10.5281/zenodo.19696191>
27. Klimis, V., Bensoussan, A., Chachkarova, E., Even-Mendoza, K., Fortz, S., Lenihan, C.: Shaking up quantum simulators with fuzzing and rigour. *Proc. ACM Program. Lang.* **9**(OOPSLA2) (2025). <https://doi.org/10.1145/3763100>
28. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: *CAV 2013* (2013). https://doi.org/10.1007/978-3-642-39799-8_1
29. Kunčar, O.: Correctness of Isabelle’s Cyclicity Checker: Implementability of Overloading in Proof Assistants. In: *Proceedings of the 2015 Conference on Certified Programs and Proofs* (2015). <https://doi.org/10.1145/2676724.2693175>
30. Livinskii, V., Babokin, D., Regehr, J.: Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* **4**(OOPSLA) (2020). <https://doi.org/10.1145/3428264>
31. Mao, K., Harman, M., Jia, Y.: Sapienz: multi-objective automated testing for android applications. In: *ISSTA 2016*. p. 94–105. ACM (2016). <https://doi.org/10.1145/2931037.2931054>
32. Meng, R., Mirchev, M., Böhme, M., Roychoudhury, A.: Large language model guided protocol fuzzing. vol. 2024 (2024), <https://dx.doi.org/10.14722/ndss.2024.24556>
33. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of unix utilities. *Commun. ACM* **33**(12), 32–44 (1990). <https://doi.org/10.1145/96267.96279>
34. Milner, R., Harper, R., MacQueen, D., Tofte, M.: *The Definition of Standard ML*. The MIT Press, 2 edn. (1997). <https://doi.org/10.7551/mitpress/2319.001.0001>
35. Owens, S., Reppy, J., Turon, A.: Regular-expression derivatives re-examined. *Journal of Functional Programming* **19**(2), 173–190 (2009). <https://doi.org/10.1017/S0956796808007090>

36. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: IWIL-2010: 8th International Workshop on the Implementation of Logics. EPiC Series in Computing, vol. 2, pp. 1–11. EasyChair (2012)
37. Pham, V.T., Böhme, M., Roychoudhury, A.: AFLNET: A greybox fuzzer for network protocols. In: ICST. pp. 460–465 (2020). <https://doi.org/10.1109/ICST46399.2020.00062>
38. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for c compiler bugs. In: PLDI '12. p. 335–346. ACM (2012). <https://doi.org/10.1145/2254064.2254104>
39. Schulz, S.: System description: E 1.8. In: LPAR-19 (2013). https://doi.org/10.1007/978-3-642-45221-5_49
40. Sulzmann, M., Lu, K.Z.M.: POSIX Regular Expression Parsing with Derivatives. In: Functional and Logic Programming, vol. 8475, pp. 203–220. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07151-0_13
41. Sutcliffe, G.: The TPTP problem library and associated infrastructure: From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning* **59**(4), 483–502 (2017). <https://doi.org/10.1007/s10817-017-9407-7>
42. Wiedijk, F.: Pollack-inconsistency. *Electronic Notes in Theoretical Computer Science* (2012), <https://www.sciencedirect.com/science/article/pii/S157106611200028X>
43. Xia, C.S., Paltenghi, M., Le Tian, J., Pradel, M., Zhang, L.: Fuzz4All: Universal fuzzing with large language models. In: ICSE '24. ICSE '24, ACM (2024). <https://doi.org/10.1145/3597503.3639121>
44. Xu, J.: Private communication: bug report on `ccl_clauses.c` with root-cause analysis and fix proposal (2026)
45. Xu, J.: Source-Informed Fuzzing of Sledgehammer’s External ATP Interface (2026). <https://doi.org/10.5281/zenodo.19899187>
46. Yang, C., Deng, Y., Lu, R., Yao, J., Liu, J., Jabbarvand, R., Zhang, L.: WhiteFox: White-box compiler fuzzing empowered by large language models. *Proc. ACM Program. Lang.* **8**(OOPSLA2) (2024). <https://doi.org/10.1145/3689736>
47. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. pp. 283–294. ACM (2011). <https://doi.org/10.1145/1993498.1993532>
48. Yu, J., Lin, X., Yu, Z., Xing, X.: LLM-Fuzzer: Scaling assessment of large language model jailbreaks. In: USENIX Security. pp. 4657–4674. USENIX Assoc. (2024), <https://www.usenix.org/conference/usenixsecurity24/presentation/yu-jiahao>
49. Zalewski Michal: Technical “whitepaper” for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt (Retrieved April 21, 2023)

A Additional Results

We discuss an additional, less impactful finding of §4.1.

Finding #5: Asterisk as reserved word. ML97 does not list '*' as a reserved word; it should be a valid symbolic identifier. Poly/ML assigns '*' a dedicated **Asterisk** token type, handling it alongside genuinely reserved symbols. This is likely a deliberate deviation to simplify parsing of tuple type expressions (e.g., `int * string`). ■

We discuss an additional finding of §4.2.

Finding #7: Lexer unbounded memory allocation on pathological floating-point literals. The `readChars` function in `LEX_.ML` reads exponent digits in a floating-point literal with no bound on accumulation length: allocation scales from 20 MB at 200 digits to 192 MB at 1,000,000 digits:

```
1 [Input]
2 5e~[10000 digit characters]
3
4 [Expected]
5 Lexer error or bounded allocation
6
7 [Observed]
8 Memory exhaustion (unbounded allocation; OOM during campaign)
```

This issue was discovered during fully automated Phase 1 greybox fuzzing. It is unclear whether this constitutes a defect: the SML Definition does not bound the exponent length, and the threshold beyond which further digits carry no semantic effect is a property of IEEE 754 arithmetic rather than a lexical one. Yet, to avoid the ambiguity, we wrote a local fix, thus (`readCharsMax`) caps accumulation at 20 characters and emits a lexer error on overflow. We reported it as Issue #273 (previously unknown, unconfirmed). ■

We discuss additional findings of §4.3 from Part-1.

Finding #10: Progress.stopped watchdog corruption. The `stopped` property on the `Progress` class (`src/Pure/System/progress.scala`) used `Thread.interrupted()` to read the thread's interrupt flag. `Thread.interrupted()` consumes the flag as a side effect, and the implementation cached the consumed value, so any unrelated interrupt (e.g. from a watchdog timer or a cancelled future) was permanently stored as a cancellation state even when `stop()` was never called:

```
1 [Caused By]
2 progress.bash() invoked with watchdog = 1 ms on a fast command
3
4 [Expected]
5 stopped = false (no stop() call was issued)
6
7 [Observed]
8 stopped = true (interrupt flag misclassified as cancellation)
```

A whitebox test using a 1 ms watchdog on a fast command triggered this behaviour at iteration 117 of 10,000. Replacing the implementation with: `def stopped: Boolean = is_stopped` eliminated the failure across 10,000 iterations. An identical fix was later found in the development branch; the report is included to confirm the LLM-assisted analysis matches expert review.

Finding #11: XML serialisation design flaw: Recursion Depth Risk. LLM-assisted scanning of `src/Pure/PIDE/xml.scala` identified robustness-related issue, in which each nesting level in the recursive traversal in `filter_elements`

(strips internal markup before display) adds one JVM stack frame. Thus, deeply nested proof terms produced by automated tactics can exhaust the default thread stack, throwing `StackOverflowError` from the serialisation pipeline. This issue does not appear in normal proof workloads, but when triggered, it produces silent corruption or session loss, making it harder to diagnose than a clear failure. A patched version reimplemented `filter_elements` iteratively (handling 500,000 levels without issue), was submitted and acknowledged by the developers

Finding #12: XML serialisation design flaw: null handling corruption. Similar to finding #11, but in the null handling in `XML.Output.string`. Here, a defensive `null` check exists, but the null branch executes `builder += str`, which delegates to `StringBuilder.append(null: String)`; the JDK specification for that call appends the four characters `n, u, l, l` rather than producing empty output, silently injecting the literal string `null` into serialised XML. As in finding #11, we have disclosed this finding via Isabelle’s mail list and submitted a patch making the null branch a no-op to the developers, who acknowledged the patch ■

We discuss additional findings of §4.3 from Part-2.

Finding #14: Cascade of intermediate proof-step markers. A single broken tactic in `Median_Of_Medians_Selection.thy` (a duplicated by `(auto ...)` step) produced one terminal error in the batch oracle but seven distinct error markers in the PIDE server, on the surrounding intermediate steps that depended on the broken context. *User impact:* the IDE displays a flood of unrelated red markers, making it difficult to locate the actual cause amongst the propagated noise.

Finding #15: Off-by-one line attribution. Across five iterations spanning two distinct theories, the server attributed an error to the line preceding the failing command while the oracle attributed it to the failing command itself. In `Roy_Floyd_Warshall.thy`, the oracle pointed at line 79 (an `obtain` command) while the server pointed at line 78 (the preceding `by sorry`). *User impact:* the IDE highlights the wrong line, sending the user to investigate code that is not at fault.

Finding #16: Differing attribution depth in nested proofs. A failure inside a nested `qed` block in `GenClock.thy` was attributed by the server to the innermost `qed` (line 431) and by the oracle to the enclosing `qed` (line 455) — a 24-line gap. Both are defensible, but they describe the same failure differently. *User impact:* A user following the IDE’s pointer is taken to a different structural level than a user reading the batch log, complicating cross-tool debugging.

Finding #17: Misleading syntax error on `by sorry` substitution. The most consistently reproducible divergence: replacing `by simp` with `by sorry` in a structured proof produces a `Bad context for command qed` message in the oracle (at the `qed` line) and an `Outer syntax error: keyword “(” expected in the PIDE server` (at the `by sorry` line). A dedicated reproducer triggered the divergence in 10/10 runs. The finding was disclosed to the Isabelle developers’ mailing list; Fabian Huch and Makarius Wenzel attributed it to the historically separate evolution of batch and PIDE error-reporting paths and indicated that

no patch was planned. *User impact*: the IDE reports a syntax error where the actual problem is a missing proof, misdirecting the user’s attention.

Finding #18: First-error truncation. A theory containing two independent proof failures was reported in full by the batch oracle but only partially by the PIDE server, which surfaced only the first failure. *User impact*: on a large theory with multiple independent lemmas, the user must fix the first error and re-run before subsequent failures become visible, producing an iterative cycle the batch build avoids.

Finding #19: Raw ML exception surfacing to the user interface. Three iterations of `Birkhoff_Finite_Distributive_Lattices.thy` produced³ reaching the IDE rather than a clean Isabelle error message. The `OF` combinator raises a raw THM exception when unification fails, and the call site does not guard against this with `Exn.capture`, so an internal ML-level stack trace propagates through the PIDE protocol and is rendered verbatim in jEdit. *User impact*: the user is shown an internal implementation detail (a file name and line number from `drule.ML`) instead of a user-facing diagnostic, undermining the abstraction that the kernel’s internals are not part of the user’s interface. ■

We discuss additional findings of §4.4 from Part-1.

Finding #21 `CCOption.get_exn` crash in higher-order unifier. Replaying the TPTP benchmark `ANA088~1.p` against Zipperposition reliably produced an unhandled `CCOption.get_exn` failure originating in `Unif.ml`, function `restrict_fun2.mk_rhs`. Root-cause analysis traced the failure to a mismatch between two notions of de Bruijn index equality: `T.is_bvar_i`, which compares index values, and `T.equal`, which compares the index plus type. On polymorphic THF inputs, the same de Bruijn index can have different types across scopes, so `T.equal` fails and the unconditional `get_exn` crashes.

```
1 [Input]
2 ANA088~1.p (TPTP problem library, polymorphic THF)
3
4 [Expected]
5 Theorem (Vampire 4.8 returns Theorem in 0.053 s).
6
7 [Observed]
8 Fatal exception: CCOption.get_exn (Unif.ml).
```

We reported this finding as Zipperposition issue #104, along with pull request #105 and pull request #106. This was acknowledged by the maintainers (confirmed, fixed, closed). The `bisect_ppx` coverage data accompanying the report showed that the failing branch was untouched by Zipperposition’s existing test suite, accounting for the defect’s survival. ■

We discuss additional findings of §4.4 from Part-2.

Finding #23: Code-quality observation: `role_of_string` (not a bug on the user path). The `role_of_string` helper in Zipperposition’s TPTP frontend uses `failwith` as the catch-all for unrecognised TPTP roles (e.g. `corollary`, `interpretation`, `logic`), which raises an uncaught OCaml exception with no source location. Sledgehammer never produces these roles; E and Vampire reject identical inputs

³ exception THM 0 raised (line 307 of "drule.ML"): OF: no unifiers

with graceful, location-bearing parse errors, so the user-visible outcome is the same. This is therefore an error-handling stylistic difference rather than a defect on the supported call path. We reported it as Zipperposition issue #108, with pull-request #109 (confirmed, with some discussions taking place in PR 109 before fix). ■

B Code Coverage of Sledgehammer’s Interface to External ATPs Seeds

We report here the Sledgehammer-related experiments, which correspond to Part of the overall experimental population in §4.4.

Coverage results: `bisect_ppx` for Zipperposition raised line coverage from 19.59% (default test suite) to 27.57% under the targeted corpus, with `Unif.ml` (the site of Finding #21 above) climbing by 30.4 percentage points; `gcc/lcov` for E prover showed baseline 26.9% versus 3.2% on the Sledgehammer-driven path, with `ccl_clauses.c` (the site of Finding #22 above) at 33.0% versus 0.4%—an 82.5× disparity. The under-exercising of `ccl_clauses.c` on the user path explains how a five-line crash survived in shipping software for several minor releases.