Ahead-of-time Analysis of Shell Program Effects

Lukas Lazarek
Brown University
USA

Evangelos Lamprou
Brown University
USA

George Kapetanakis
Brown University
USA

Eric Zhao
Brown University
USA

Zhiwen Zheng
Brown University
USA

Michael Greenberg
Stevens Institute of Technology
USA

Konstantinos Kallas
UCLA
USA

Nikos Vasilakis
Brown University
USA

Abstract

The Unix shell remains a core system substrate across system administration, automation, and software development. Shell programs, however, are prone to subtle, severe, and often irreversible effects that are difficult to predict. The challenge stems from the shell's unique execution model, its reliance on external computation and state, its highly dynamic expansion semantics, and the complex interactions among these features. This paper presents SaSh, a system that statically analyzes shell programs to identify errors in their execution before they occur. SaSh introduces an optimistic symbolic execution engine for shell programs that limits path explosion and focuses on high-impact failures. It tracks the effects of external commands over a filesystem model, and approximates shell word expansion using a tailored abstract domain. SaSh quickly identifies bugs even in large programs with a risk-directed exploration strategy, steering its analysis to program fragments likely to exhibit dangerous behavior. Applied to 61 buggy programs, including several high-profile disasters, SaSh identifies all but one instance of unwanted behavior with no false positives, going far beyond the current state-of-the-art. Furthermore, SaSh has already yielded 70 bug reports in 44 open-source projects such as PyTorch, the P4 Compiler, Kubernetes, and vLLM, including bugs that can lead to irreversible data loss.

1 Introduction

The shell is one of the most prevalent programming environments [57, 97, 107], supporting a variety of system management and automation tasks—from software installation, to data management, to one-off filesystem reorganizations—used by a broad spectrum of developers, from engineers at large software companies to everyday non-specialists [144] and, increasingly, generative AI agents [65]. It allows manipulating data and system resources using dozens of built-in constructs, a rich set of standard utilities, and third-party commands. Its dynamism, as reflected in features such as word expansion and command invocation, allows developers to succinctly express complex workflows that query and respond to the environment, adapt to varying contexts, and handle user inputs.

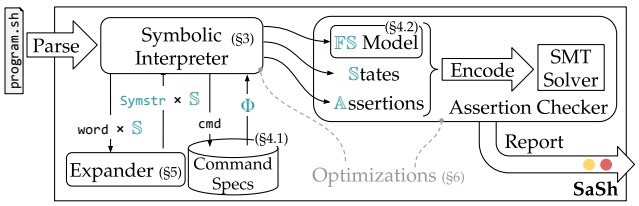


Fig. 1. High-level overview of SaSh. SaSh analyzes shell programs ahead of their execution, identifying bugs before they occur.

Unfortunately, these features also create potential for catastrophic bugs, and stymie rich program analyses that could warn developers about them beforehand. External commands execute arbitrary, opaque binaries to perform much of a shell program's actual work; expansion dynamically determines core elements of the program (including which commands to run) as it executes; and all of these features depend upon, query, and modify the surrounding environment pervasively. Each of these features poses a significant challenge for ahead-of-execution program analysis; but their combination can be disastrous: a 2015 bug in Valve's Steam updater deleted entire filesystems [134], a 2001 bug in Apple's iTunes updater wiped out user drives [135], and a 2011 bug in the open-source Bumblebee installer for Nvidia drivers rendered systems unbootable by deleting `/usr` [131].

This paper presents SaSh, a system for automatically detecting such catastrophic bugs deep in shell programs *ahead of time*, notifying their developers *during* development and their users *before* their execution—not *after* disaster has occurred (Fig. 1). SaSh's analysis reasons at a much deeper level than current state-of-the-art systems [25, 33, 71] using symbolic execution: it simulates program execution *symbolically* according to the shell's semantics, and reasons about program effects with *approximations* over a *model* of the system environment. Instead of concrete values for inputs and environment state, SaSh manipulates symbolic representations describing all possible values, inputs, and environments.

```
1 #!/bin/sh
2 STEAMROOT="$(cd "${0%/*}" && echo $PWD)"
3 # ... 342 more lines ...
4
5 rm -rf "$STEAMROOT/"*
6
```

Fig. 2. The core of the Steam updater bug [134]. When expansion results in an empty `STEAMROOT` string (ln. 2), the program deletes everything user-writable (ln. 5).

During analysis, SaSh collects constraints on these symbolic values and leverages a solver to determine the feasibility of potentially dangerous behavior expressed as *semantic assertions* over the symbolic state. To reason about latent dangers in the shell's dynamic, word-expansion semantics, SaSh assigns symbolic values to an *abstract expansion domain* capturing the range of possible expansions every value may result in; this abstract domain uncovers expansion-related bugs efficiently, despite the mechanism's complex dynamism.

To reason about opaque commands, SaSh employs concise *specifications* that summarize command behavior and effects. These specifications query and manipulate the symbolic shell environment and filesystem model. By composing these specifications according to the program's semantics, SaSh identifies behaviors that may lead to catastrophic consequences—from both individual command invocations and their compositions.

Additional optimizations allow SaSh to identify bugs deep inside large and complex programs. Foremost among them, *risk-directed exploration* allows SaSh to focus on program fragments more likely to lead to dangerous behavior, by considering the potential effects of commands inside them.

SaSh has been evaluated on a collection of 116 documented bugs across 61 real-world shell programs from a variety of sources, including StackOverflow, GitHub, the Debian bug-tracking system, and bugs from Steam, iTunes, Nvidia, Ubuntu, and others. SaSh identifies 115 out of 116 bugs in these programs: 97/116 (83.6%) within one second, and 115/116 (99.1%) within 60s. It reports *no* false positives on the fixed versions of these programs. SaSh goes far beyond the current industry-standard shell linter, which reports 42 (36.2%) of these bugs—easily thwarted by trivial syntactic variations. Optimizations increase SaSh's bug-finding performance by 18% and allow it to identify bugs deep inside large programs within 60s that the un-optimized system cannot find even within 1h. SaSh has also discovered 70 *previously unknown bugs* in open-source systems, including PyTorch, Next.js, the P4 compiler, Kubernetes, and vLLM, all of which have been reported to their maintainers.

The paper begins by applying SaSh to the Steam disaster (§2). It then contributes:

- **An optimistic symbolic execution engine** (§3). SaSh simulates the execution of shell programs using symbolic variables that represent many possible values and environments at once, and incorporates knowledge of important domain-specific failure modes to simplify and prune the symbolic state space across multiple dimensions.
- **Effect and environment modeling** (§4). It introduces a non-hierarchical filesystem model that enables efficient reasoning about program-environment interactions, focusing only on paths relevant to program behavior. SaSh models command behavior with compositional pre- and post-condition specifications over a symbolic environment.
- **Abstract expansion domain** (§5). SaSh maps the shell's expansion constructs to an abstract domain that captures coarse constraints on each expanded field's content and arity, allowing SaSh to reason about disaster-prone expansions without explicitly considering all possible outcomes.
- **Risk-directed path prioritization** (§6). Prior knowledge about potential command effects allows SaSh to employ several domain-specific optimizations, such as path prioritization for potentially catastrophic executions. Additional optimizations include path merging, pruning, and in-flight simplifications that reduce solving overhead.

The paper then reports on SaSh's evaluation (§7), discusses related work (§8), and finally concludes (§9). Appendices provide additional details on SaSh's evaluation set (App. A), newly discovered bugs (App. B), and abstract expansion process (App. C).

Availability: SaSh will be made available as MIT-licensed software, alongside all evaluation data at:

https://github.com/atlas-brown/sash

2 Example & Overview

This section presents a real disaster caused by a bug in a shell script, and walks through SaSh's successful analysis.

A disastrous bug: Fig. 2 presents a snippet from the Linux Steam updater [17], intended to identify and clear its parent directory (`$STEAMROOT`) before installing a new version of Steam. Its `cd` inside a subshell changes directory into the path pointed to by `${0%/*}`. This expansion removes the shortest suffix starting with a slash (`/`) from `$0`, which corresponds to the program's invocation path (`/opt/steam/update.sh`). The `cd` then enters the directory (*e.g.*, `/opt/steam`); if it succeeds (`&&`), `echo` outputs `$PWD`—the current directory just entered. Running all this inside a command substitution (`$()`) means `echo`'s output is captured in `STEAMROOT` and the change in directory does not propagate to the rest of the script. Finally, `"$STEAMROOT"/*` expands to `/opt/steam`'s top-level entries, and `rm -rf` deletes them all, recursively.

Unfortunately, when directly invoked from its installation directory (with no path qualifiers), `$0` becomes `update.sh` and thus `${0%/*}` expands to `"update.sh"`, `cd` fails, preventing `&& echo $PWD` from executing and producing no output. The resulting expansion, `""`, is assigned to `$STEAMROOT`,

```
1 #!/bin/sh
2 STEAMROOT="$(cd "${0%/*}" && echo $PWD)"
3 # ... 342 more lines ...
4 if [ "$STEAMROOT" != "" ]; then
5     rm -rf "$STEAMROOT/"*
6 fi
```

Fig. 3. An obviously safe fix to the Steam bug (*Cf.* Fig. 2). The `rm -fr` will *never* delete from the root—across all executions and environments—yet ShellCheck still warns that it could.

Steam's `update.sh`	Analysis steps	
2 `STEAMROOT="`	$\texttt{STEAMROOT} = x$	①
2 `  $(cd "${0%/*}"`	`${0\%/*}` ⇢ dir	②
2 `  &&`	$\$?_{\texttt{cd}} = 0 \leftarrow \textit{fork} \rightarrow \$?_{\texttt{cd}} \neq 0$	③
2 `  echo $PWD)"`	$x = \texttt{\$PWD}$; $x = \texttt{""}$	④
5 `rm -rf "$STEAMROOT/"*`	$\texttt{"}x\texttt{/"*}$ ⇢ del ; `/*` ⇢ del	⑤

Fig. 4. Parts of SaSh's analysis of the Steam bug (*Cf.* Fig. 2). The failed `cd` path yields `STEAMROOT=""`, so the final `rm` targets `/*`.

which in turn causes the last line to expand to `rm -rf /*`, deleting everything user-writable.

Challenges: Detecting such bugs in shell programs ahead-of-time is challenging. ShellCheck [33], the *de facto* shell linter used by over 60% of popular shell repositories on GitHub, would not identify this bug at the time of its occurrence [132], as no syntactic lint for this circumstance was implemented. Even ShellCheck's latest version (v0.11.0) is brittle: it fails to detect a small, semantically-equivalent variation of the bug—moving the final slash from the `rm` command (ln. 5) to the end of the `STEAMROOT` assignment (ln. 2)—and mistakenly warns about Fig. 3's fix. Reliable analysis demands deeper reasoning about shell programs, entailing several challenges.

The first challenge (C_1) is handling the shell's unique execution model. Almost every command in a shell program might fail, and most invocations that interact with external values could, in principle, be catastrophic. For instance, a malicious invocation may cause the Steam updater's `"${0%/*}"` to expand to `/home`, causing `rm -rf` to delete it. However, considering such malicious scenarios amounts to warning on the vast majority of practical `rm` invocations. Avoiding this noise requires an analysis that identifies failures that occur even under *optimistic* assumptions: these are high-impact bugs that affect everyday usage under common conditions.

A second challenge (C_2) when analyzing shell programs is reasoning about both the wider environment and opaque commands that interact with it. For example, Fig. 2's `cd` invocation succeeds only when the given argument points to a directory. Even seemingly small, self-contained units of code regularly depend on and affect global state, making purely local reasoning insufficient. At the same time, shell programs rely heavily on external commands—standalone executables wholly distinct from the program under analysis. To reason about overall program behavior, a system must be able to reason about the behavior of these commands on the shell state and the system environment in a way that is both precise, to identify bugs, and tractable, to be applied at scale.

A third challenge (C_3) is reasoning about the possible outcomes of shell expansion. For example, Fig. 2's bug manifests by the interplay of *four* different types of expansion. Explicitly enumerating all possible outcomes of this complex, dynamic process is infeasible. Practical analysis *must* make a sound approximation of the shell's expansion semantics.

The last challenge (C_4) is handling the exponential growth of execution paths induced by the shell language's density even on moderately-sized programs. Every command implicitly introduces a branch depending on whether it succeeds or fails, which, when combined with other shell constructs such as word expansion, grows the number of feasible paths rapidly. The full Steam updater program is hundreds of lines long, making complete path exploration infeasible. An effective bug-finding analysis must therefore focus exploration on paths most relevant for actionable bug-finding.

SaSh's approach: Invoking SaSh on Steam's `update.sh` program with a five-second time budget identifies the bug:

```
$ sash --timeout 5 /opt/steam/update.sh
> Line 359: Deletion of /* due to empty $STEAMROOT.
> ... more warnings
```

Internally, SaSh starts by parsing the script and initializing the symbolic shell environment state, mapping all environment variables defined by the POSIX standard [60] with unconstrained symbolic values (§3.2). It also initializes its symbolic filesystem as an associative map from paths to objects, each modeled as either a file, a directory, deleted, or an unknown state (§4.2).

SaSh then starts symbolically executing the script at the first executable line, which assigns the result of the command substitution to the `STEAMROOT` variable (① in Fig. 4). The engine enters the substitution and evaluates it. On the operator's left side, it interprets the `cd` invocation symbolically (②), including the expansion `${0%/*}`, modeled as an unconstrained environment-derived string ($C_{2,3}$) (§4.1–§5). At the outer level, the `&&` operator then introduces control-flow branching (③): for each command whose success affects the conjunction ($C_{1,4}$), SaSh forks the symbolic state to represent both the success and failure paths. Because `cd` is a shell built-in, SaSh implements its behavior according to POSIX (§3.2): It updates the current working directory `$PWD` to the path given by `${0%/*}`, asserts that it must be a directory (to be checked later), and updates `$OLDPWD`.

Next, along the subshell's success path, SaSh evaluates `echo` (④), another built-in which deterministically succeeds along the success branch of `cd`, outputting `$PWD`. The assignment binds this value to a fresh symbolic `STEAMROOT` variable, and SaSh constrains $\texttt{\$STEAMROOT} = \texttt{\$PWD}$, where `$PWD` is the subshell's working directory after `cd`.

Along the subshell's failure path, the `cd` fails with no output and SaSh adds the constraint `$STEAMROOT = ""`.

Once SaSh reaches the `rm` invocation (⑤), it symbolically expands the `"$STEAMROOT/"*` argument, which is composed of three parts: the symbolic variable `STEAMROOT`, a literal slash, and the `*` glob (§5). For each part, SaSh lowers it to an abstract expansion domain that captures the range of words it may expand to: `STEAMROOT` may expand to zero or more words, the slash is constant, followed by `*`, which can expand to zero or more words (C_3). SaSh then fetches `rm`'s specification (§4.1): it states that all paths specified by the arguments are deleted (C_2). SaSh adds those facts to the execution path's path condition (§3.2) and applies them to the symbolic filesystem (§4.2).

Observing the deletion in the postcondition, SaSh discharges a semantic assertion stating that none of the deleted paths are critical (including everything under `/`) (⑤, failed `cd` path). It then asks the solver whether the accumulated path conditions are satisfiable and whether the assertion is satisfiable. The solver confirms this is *not* the case for the path where `cd` fails, and SaSh reports the bug.

From 2 to 450 lines: The full updater program contains significantly more possible execution paths. The two statements are 342 lines apart, with 83 explicit branching points until the dangerous `rm`. This complexity makes complete path exploration infeasible, thus SaSh incorporates two domain-specific path-management strategies ($C_{1,4}$). First, it optimistically assumes commands succeed outside explicit control flow (§3.2); it also prioritizes paths more likely to lead to catastrophic consequences, accelerating bug identification even in large programs, before thoroughly exploring them (§6).

As a result, SaSh detects and warns about the bug in the full Steam updater program (450 lines of code) in 4.8s, while producing no such warning on the fixed version sketched in Fig. 3 with a similar detection time. Meanwhile, ShellCheck v0.3.5 (from 2015) produces no relevant warning in either version, and v0.11.0 produces the exact same warning for both versions (it blindly warns about the `rm` invocation pattern).

3 SaSh Core

SaSh explores program behaviors to surface executions manifesting bugs (§3.1) using symbolic execution (§3.2–§3.3).

3.1 What Counts as a Bug?

The shell is highly permissive with respect to errors: the kinds of execution states that result in failure in other environments—*e.g.*, commands failing, syntax errors, and missing dependencies—often do not halt shell program execution nor result in immediately observable exceptional behavior. This offers shell developers fine-grained control [59], but also opens the door for unexpected behavior. Hence, the vast majority of real-world bugs in shell programs are *misbehaviors*: perfectly normal executions according to the shell's semantics but with unexpected or undesired effects.

Table 1. Classes of misbehavior. Each row describes a class of bug patterns SaSh detects, with a concrete example manifestation.

Description	Example
Delete critical path	`rm -rf "$HOME"`
Dangerous word split	`rm -rf /usr/$1`
Data loss	`mv a x.txt; mv b x.txt`
Command will fail	`rm a.txt; cat a.txt`
IO mismatch	`VAR=$(mv a b)`
Missing command	`command -v foo || foo abc`
Identifier misuse	`cd "$UNBOUND"; cmd > fun`
Bad control	*Const. cond., infinite loop,* etc.

SaSh defines eight classes of misbehaviors (Tab. 1) to identify as bugs, derived from a review of its benchmarks that yielded a taxonomy of common failure classes (*Cf.* §7). These misbehavior classes differ fundamentally from linters (*e.g.*, ShellCheck)—which check for syntax patterns—because they are not about syntax, but rather about the semantics of possible program executions. The first class captures bugs that cause deletion of critical paths (*e.g.*, `/`, `/home`) and is what triggers SaSh's warning in the Steam updater example (§2). The next five classes capture unwanted behavior stemming from command compositions and their interplay with shell expansion. The seventh class (identifier misuse) covers both the use of unbound variables and incorrect references to functions. The last class captures patterns sometimes used intentionally—such as infinite loops—but are common sources of bugs.

SaSh is parameterized over these misbehavior classes and, for each, derives semantic assertions. SaSh implements these assertions as logical constraints and reports their violations as potential bugs.

3.2 SaSh Overview

SaSh's symbolic execution engine interprets shell programs over *symbolic variables* representing unknown inputs and environment state, accumulating constraints and semantic assertions as execution proceeds, and finally checking the satisfiability of all collected assertions. Central to it is an optimistic execution strategy tailored for impactful bug-finding.

Optimistic bug finding: Fundamentally, most computation inside shell programs is delegated to external commands. This can make their analysis both intractable and uninformative. Intractable because commands can fail in many fine-grained ways: `rm a b c`, for example, may fail to delete *any* subset of its arguments, so even short command sequences induce combinatorial state explosion. Unhelpful because of their pervasive environment dependence and interaction:

Symbolic interpreter state is one of each:

$$\mathbb{S}\left[\begin{array}{rrcl} \text{Path condition} & \mathbb{P} & ::= & \overline{\mathbb{C}} \\ \text{Assertions} & \mathbb{A} & ::= & \overline{\mathbb{C}} \\ \text{Filesystem} & \mathbb{FS} & & (\textit{Cf.}\ §4.2) \\ \text{Terminated?} & \mathit{term}? & ::= & T \mid F \\ \text{Global env.} & \rho_g & : & \mathit{str} \mapsto \text{Symstr} \\ \text{Local env.} & \rho_\ell & : & \mathit{str} \mapsto \text{Symstr} \\ \text{Function env.} & \rho_f & : & \mathit{str} \mapsto \text{AST} \\ \text{Exit status} & \mathit{ES} & ::= & (0 \mid \neg 0) \times \mathit{known}? \\ \text{Interp. config} & \mathit{opts} & \subseteq & \{\texttt{allexport}, \ldots\} \end{array}\right] \mathsf{S}$$

Strings	str		
Symbolic strings	Symstr		(*Cf.* §5)
Known?	$\mathit{known}?$	::=	$T \mid F$
Constraints	$\mathbb{C}$		(*Cf.* §4.1)

Fig. 5. Symbolic interpreter state. The symbolic state $\mathbb{S}$ consists of nine left-bracketed fields. The four right-bracketed fields make up the modeled shell state S. Symbolic fields are marked.

almost any script admits a dangerous execution in *some* pathological environment, so warning on mere possibility of failure would flag nearly every part of any program.

SaSh avoids both concerns with *optimistic symbolic execution*. First, the system executes commands optimistically: SaSh considers that a command either succeeds or fails as a whole, and it explores failure only for explicit conditional control. Second, SaSh reasons about overall behavior optimistically: rather than speculating about any possible failure mode, it focuses on identifying bugs that manifest even under the ideal environment defined by the successful execution of the program up to any given point. In other words, SaSh identifies when the program's behavior is internally inconsistent, or dangerous even under ideal conditions. This lets the analysis identify *real* bugs with high signal-to-noise, flagging faults unambiguously attributable to the program.

Overview: The system comprises three core components: the interpreter, expander, and assertion checker (Fig. 6).

SaSh implements a tree-walking symbolic interpreter for shell programs; it maps a collection of symbolic interpreter states and a shell program to a new collection of symbolic states. As part of evaluation, interpretation is interleaved with expansion, which transforms AST-level strings (words) into symbolic strings, each paired with a resulting state (§5). For instance, interpreting `$(echo "$PWD")` involves expanding the command substitution, going back into interpretation to interpret `echo`, and then returning to expansion to produce the command substitution's final value.

Both interpretation and expansion produce semantic assertions, which are constraints over the symbolic state that, if violated, indicate a bug (§3.1). Finally, after interpreting a program, SaSh uses a solver to analyze these assertions and reports if any are unsatisfiable.

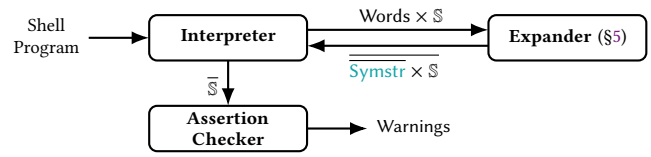


Fig. 6. Symbolic execution components. The interpreter and expander are mutually recursive: interpretation triggers expansion for word evaluation, and expansion invokes interpretation for command substitution. The assertion checker then analyzes the resulting constraints and warns on unsatisfiability.

3.3 Symbolic Execution

After parsing a shell program into an AST, SaSh initializes a state with the standard environment variables defined by POSIX (*e.g.*, `HOME`, `PWD`) as fresh symbolic variables, and scans the program AST to discover all function definition nodes (in order to distinguish not-yet-defined functions from unknown commands). It leaves all other fields in an initial (empty) state. SaSh then interprets the program starting with that single state.

Symbolic state: SaSh represents a single point of execution using a symbolic interpreter state $\mathbb{S}$. This state consists of nine central elements, which suffice for it to model the shell behaviors it needs to find bugs (Fig. 5). The path condition $\mathbb{P}$ describes constraints on the execution path represented by each state. Semantic assertions $\mathbb{A}$ are a set of constraints to check, corresponding to bugs in the program when unsatisfiable. The symbolic filesystem model $\mathbb{FS}$ tracks the state of paths manipulated by the program (§4.2). The global and local variable environments ρ_g and ρ_ℓ map identifiers to symbolic strings and the function definition environment ρ_f maps identifiers to function body AST nodes. The exit status *ES* (corresponding to `$?`) tracks the latest exit status together with a *known*? flag indicating whether that status is statically known (vs. optimistically assumed). The termination status *term*? tracks whether the path represented by the state has terminated. Finally, the interpreter configuration *opts* tracks active shell interpreter configuration options as manipulated by `set`.

Symbolic interpretation: The interpreter treats several classes of shell constructs differently. For core shell constructs such as assignment and sequencing, SaSh reimplements the standard concrete interpreter to operate over symbolic values. SaSh implements POSIX builtins [61] directly but operating over symbolic values. For command invocations not modeled directly, it uses concise specifications that describe the invocation's expectations on the state (whose violation corresponds to a bug) and the effects of the invocation on the state (§4.1).

SaSh explores both outcomes of branching constructs such as **`if`**, `&&`, `||`, `!`, and **`case`**, and then merges the resulting

Filesystem	$\mathbb{FS}$	:	$p \mapsto s$
Constraints	$\mathbb{C}$	::=	$p \dashrightarrow s \mid \mathsf{c}_?(p) \mid io$
			$\mid \mathsf{Symstr} = \mathsf{Symstr} \mid \mathbb{C} \Rightarrow \mathbb{C}$
			$\mid \neg\mathbb{C} \mid \mathbb{C} \wedge \mathbb{C} \mid \mathbb{C} \vee \mathbb{C} \mid \top$
Specifications	Φ	:	$\mathsf{AST} \to (\phi_a \twoheadrightarrow \langle \mathbb{C}_s,\ \mathbb{C}_f \rangle)$
Path state	s	::=	$(\mathsf{file} \mid \mathsf{dir} \mid \mathsf{del}) \times \mathit{read}?$
Output	io	::=	$\mathtt{stdin} \mid \mathtt{stdout}(\mathsf{Symstr})$
Paths	p	::=	Symstr
Read?	$\mathit{read}?$	::=	$T \mid F$

Fig. 7. Filesystem model, constraint and specification language. Constraints describe path states, command existence, IO information, and string equalities. Specifications are a triple of usage assertion and success or failure postconditions.

$$
\begin{array}{rcl}
[\![\mathrm{cmd}]\!] & \triangleq & \phi_a(\text{assertion}) \twoheadrightarrow \langle \mathbb{C}_s(\text{success}),\ \mathbb{C}_f(\text{failure}) \rangle \\
[\![\mathtt{cat}\ \overline{p}]\!] & \triangleq & \bigwedge_{\overline{p}} p \dashrightarrow \mathsf{file} \twoheadrightarrow \langle \bigwedge_{\overline{p}} p \dashrightarrow \mathsf{file}_r \wedge \mathtt{stdout}(x), \top \rangle \\
\mathtt{rm} \left[\begin{array}{rcl}
[\![\mathtt{rm}\ \overline{p}]\!] & \triangleq & \bigwedge_{\overline{p}} p \dashrightarrow \mathsf{file}_r \twoheadrightarrow \langle \bigwedge_{\overline{p}} p \dashrightarrow \mathsf{del}, \top \rangle \\
[\![\mathtt{rm\ \{-f,-i,-v\}}\ \overline{p}]\!] & \triangleq & [\![\mathtt{rm}\ \overline{p}]\!] \\
[\![\mathtt{rm\ \{-d,-r,-R\}}\ \overline{p}]\!] & \triangleq & \bigwedge_{\overline{p}} p \dashrightarrow (\mathsf{dir} \vee \mathsf{file}_r) \twoheadrightarrow \langle \bigwedge_{\overline{p}} p \dashrightarrow \mathsf{del}, \top \rangle
\end{array}\right. \\
[\![\mathtt{test}\ a = b]\!] & \triangleq & \top \twoheadrightarrow \langle a = b, \neg(a = b) \rangle \\
[\![\mathtt{sudo}\ c]\!] & \triangleq & [\![c]\!]
\end{array}
$$

Fig. 8. Specifications for selected command invocations. Command specifications are manually written by domain experts and capture the possible effects of each supported invocation shape on the system environment.

states, when possible (§6). For **case**, it does not record constraints induced by shell patterns, which lie outside its current constraint language. SaSh unrolls loops a configurable number of times (three by default), and fully unrolls ones with statically-determined iteratees.

SaSh executes subshells and command substitutions in a copied shell state, so it keeps shell-state changes local to the subshell (consistent with the shell's semantics) while preserving the rest of the symbolic execution state, *e.g.*, the filesystem (*Cf.* §4.2).

SaSh interprets assignments by first expanding their right-hand side and then updating each resulting state's ρ_g environment, so after `x="$STEAMROOT/"*` the binding for `x` holds the resulting symbolic string. Function definitions such as `f(){ rm "$1" }` update ρ_f, where the name `f` maps to the function's body AST.

SaSh handles function calls on a best-effort basis: if all active states agree on the same definition, SaSh interprets the body's AST subtree (otherwise treating the function call as an unknown command invocation) within environment ρ_ℓ, binding inside it positional parameters (`$1`, `$2`, *etc.*) as precisely as word splitting permits. For instance, in `f a $x c`, if `$x` may expand to multiple words, then SaSh can still bind `$1` to `a`, but abstracts `$2` and all subsequent parameters with fresh symbolic variables.

SaSh handles `eval` and `.` only when their argument is fully concrete, and treats backgrounded commands (`&`) as sequentially executed in the foreground. It omits a set of POSIX features that are orthogonal and straightforward to implement: `shift`, `times`, and most `set` options aside from `-e` and `-u`. In contrast, SaSh does not interpret `trap` at all, since `trap` code can execute at any arbitrary point in a program's execution (whenever a signal is received).

At each interpretation step, SaSh produces states paired with semantic assertions corresponding to potential bugs. SaSh emits semantic assertions at command invocations that check: (misbehavior class 1) destructive effects avoid critical paths, (2) expansions are not susceptible to dangerous splitting, (3) filesystem updates do not clobber unread data, (4) commands are not guaranteed to fail, (5) commands consume compatible I/O, and (6) commands exist. SaSh emits semantic assertions at identifier references and control constructs that check: (7) variables and functions are defined consistently, and (8) branch and loop conditions are non-degenerate (*e.g.*, constant). SaSh checks each semantic assertion by encoding the relevant path condition and filesystem state in SMT; if the assertion is unsatisfiable, SaSh reports a warning.

4 Effect and Environment Modeling

Shell programs invoke external commands that can have broad, often irreversible effects on the environment. To reason about those effects without touching the host system, SaSh simulates command effects through specifications (§4.1) over a symbolic filesystem (§4.2).

4.1 External Command Specifications

External commands are the one part of a shell program whose semantics cannot be inspected by the symbolic interpreter: after parsing and expansion, the actual shell eventually reaches `execve` and transfers control to an external executable.

To overcome this challenge, SaSh models a command's behavior as a function of its invocation as a specification triple—usage assertion, success postcondition, and failure postcondition. For example, `rm -rf "$STEAMROOT/"*`, after expansion (§5), becomes an invocation of `rm` with concrete flags `-r -f` and one symbolic operand $x_{\texttt{"\$STEAMROOT/"*}}$. SaSh assumes that flags and options are concrete, and any symbolic parameters are operands, so that it can unambiguously match the invocation to a specification. The specification matching this invocation is:

$$
[\![\mathtt{rm\ -rf\ "\$STEAMROOT"/*}]\!] \triangleq \underbrace{x_{\texttt{"\$STEAMROOT/"*}} \dashrightarrow (\mathsf{dir} \vee \mathsf{file}_r)}_{\text{usage assertion } \phi_a} \twoheadrightarrow \langle \underbrace{x_{\texttt{"\$STEAMROOT/"*}} \dashrightarrow \mathsf{del}}_{\text{success postcondition } \mathbb{C}_s},\ \underbrace{\top}_{\text{failure postcondition } \mathbb{C}_f} \rangle
$$

The usage assertion ϕ_a states that SaSh will produce a warning when it deduces that $x_{\texttt{"\$STEAMROOT/"*}}$ *cannot* be either a directory (dir) or a file that has already been read (file_r) (*i.e.*, it can only be an unread file or deleted).[1] The subscript r is the read-status bit (Fig. 7), which SaSh uses as a proxy to decide whether deleting the file risks silent data loss. This assertion, therefore, checks for both the command failure and data loss misbehaviors (*Cf.* Tab. 1). The success postcondition $\mathbb{C}_s$ states that `rm` deletes those paths. The failure postcondition $\mathbb{C}_f$ is a no-op, encoding complete failure consistent with SaSh's optimistic symbolic execution model (§3.2).

SaSh's invocation mapping function matches distinct invocation shapes to specifications; Fig. 8 illustrates its shape with representative examples, including the full set of cases for SaSh's `rm` specification. Consistent with its optimistic symbolic execution model, SaSh falls back to an empty specification for commands for which it has no specification.

In general, an invocation specification Φ is a triple $\phi_a \twoheadrightarrow \langle \mathbb{C}_s,\ \mathbb{C}_f \rangle$. The usage assertion ϕ_a is a semantic assertion that may correspond to one or more misbehavior classes, or none (*e.g.*, `test` in Fig. 8). The two postconditions describe the invocation's effects on a successful or failed execution, respectively. SaSh expresses all three components as formulas in its constraint language $\mathbb{C}$ (Fig. 7).

Beyond those illustrated by the `rm` specification, SaSh's constraint language includes additional constraint types. The predicate $\text{c}_?(p)$ expresses that a path names an existent external executable. Input-output constraints *io* describe values flowing through a command's standard input and output. Since paths are just symbolic strings, equality constraints allow SaSh to capture aliasing relationships like two different symbolic paths denoting the same location.

The current version of SaSh comes with a small, standard library of declarative command specifications that we have manually developed for several commands. Scaling this specification effort in the future will benefit from crowd-sourcing or automated specification synthesis, possibly leveraging `man` pages and natural-language processing [18, 68, 139].

4.2 Filesystem Model

SaSh uses a non-hierarchical filesystem model that maps paths to states. This encoding does not capture the filesystem's structure, but it is simple to implement, efficient to reason about, and still sufficient to find real bugs (§7). Each path state (s in Fig. 7) is a pair consisting of a kind and a status. The kind is either a *file*, *directory*, or *deleted*. The status is either *read* or *unread*.

SaSh's filesystem model interface exposes three operations: `apply`, which updates the model state from a command postcondition; `encode`, which encodes the current model state into SMT; and `assert`, which encodes an assertion to be checked against the model. This interface allows for plugging in alternate implementations that capture additional filesystem properties (*e.g.*, file permissions) in the future.

SaSh implements the filesystem as an SMT array mapping symbolic strings to path states. To implement `apply`, SaSh updates the entry keyed by the postcondition's path with the new path state that the postcondition describes. When a postcondition contains an implication, SaSh translates it into a conditional update: the premise is evaluated over the current model state, and the conclusion is applied to the subsequent state. SaSh implements `encode` by translating the flat map into an SMT array, and implements `assert` by translating each usage assertion's constraints as an SMT formula over the SMT array.

5 Abstract Expansion Domain

Expansion is a central process in the semantics of the shell, and its inherent complexity and dynamism make it a key challenge for precise symbolic reasoning. Indeed, expansion can invoke arbitrary code whose results recursively shape the program under evaluation. These expansions are challenging to reason about symbolically, given that the results are statically unknowable and unboundable.

To tame the complexity of symbolically reasoning about expansion, SaSh observes that a central source of bugs relating to expansion is the process of *field splitting*. Expansion transforms raw program text into strings, with several different mechanisms (*Cf.* Tab. 2); field splitting determines what becomes a single or multiple strings. Specifically, the shell splits up strings delimited by `IFS` characters—typically space, tab, and newline—determining how downstream executables and shell constructs see distinct strings resulting from expansion. As a concrete example, field splitting causes `X="my path/"; rm $X` to receive two distinct path arguments: `rm my path/`, deleting two different paths—a common source of bugs [26].

SaSh therefore simplifies the results of expansion down to only three properties (prefix, suffix, number of fields) that are adequate to identify bugs in a wide variety of shell scripts; it drops all other information to make reasoning more scalable. Formally, SaSh defines an abstract domain for expansion that models this information, enabling tractable, approximate expansion reasoning that is tuned for bug finding.

This section defines SaSh's abstract domain (§5.1) and how SaSh models expansion in terms of that domain (§5.2).

5.1 Abstract Domain

Fig. 9 defines SaSh's abstract domain, *symbolic strings*, for the results of expansion. A symbolic string can either be: (1) a concrete value (if it can be statically determined); or (2) a symbolic value that could represent any string, constrained by a concrete prefix and suffix ($x_{p/s}$), along with a range of the number of fields the value could be split to (FC).

[1] SaSh tracks $x_{\texttt{"\$STEAMROOT/"*}}$ as a single path in its filesystem model (*Cf.* §4.2).

Table 2. Shell word expansion mechanisms.

Mechanism	Example	Concrete	Symbolic
Tilde exp.	`~/d`	`/home/u/d`	$x_{\texttt{HOME}}$`/d`
Parameter exp.	`$v`	`abc`	x_v
Command subst.	`$(uname)`	`Linux`	x_{fresh}
Arithmetic exp.	`i=$((i+1))`	`i=42`	`i=`x_{fresh}
Field splitting	`$v`	`a b`	x_v
Pathname exp.	`*.t`	`a.t b.t`	x_{fresh}
Quote removal	`"$v"`	`"a b"`	`"`x_v`"`

Symbolic variables are augmented with concrete prefixes and suffixes to capture patterns of string concatenation common in the shell. For example, scripts often include code of the form `rm $1/usr`—meaning the concatenation of `$1`'s value with `"/usr"` (*e.g.*, the iTunes updater has code like this). Since `$1` is an unknown input to the program, SaSh represents its expansion result as an abstract symbolic variable x; however, for the words `$1/usr`, SaSh maintains the information that the expansion result has a constant component in the suffix string `"/usr"`. Similarly for prefixes.

Alongside symbolic variables, SaSh tracks the range of possible field counts the variable could be split to (*FC*). This allows SaSh to identify dangerous splits, possible critical-path deletions, and instances of bad control that depend on field splitting (*Cf.* Tab. 1). For example, in the invocation `rm $1/usr`, SaSh can identify that (absent other constraints on `$1`) field splitting of `rm`'s argument can result in deletion of unexpected paths because it can be split into multiple fields (*i.e.*, has a *FC* of $[1, \infty]$). Furthermore, with the concrete suffix, SaSh can more specifically identify at least one such unexpected splitting that could be catastrophic: if `"/usr"` splits to its own argument. While this direct invocation pattern is a common source of bugs, the field count range enables SaSh to identify field splitting bugs more generally. For instance, SaSh identifies that the following program suffers from the same problem, because the unquoted file paths may be split:

```
find ... | xargs -I{} rm {} \;
```

As another example, field counts allow SaSh to identify possible bad control resulting from field splitting, such as the following conditional that can never succeed because the output of `cat` is subject to field splitting:

```
if [ $(cat a.txt) = "a b" ];
```

5.2 Abstract Expansion

With this abstract domain, SaSh approximates the results of expansion while keeping precise track of their prefix, suffix, and field count range. SaSh handles all seven expansion mechanisms as illustrated in the Symbolic column of Tab. 2, but parameter expansion and field splitting are the most interesting ones under the abstract domain, so the rest of this section focuses on those two. Interested readers can find the rest of the expansion mechanisms in the appendix (App. C).

$$
\begin{array}{rrcl}
\text{Symbolic string} & \textsf{Symstr} & ::= & str \mid x_{p/s} \times FC \\
\text{Pre/suffixed sym. var.} & x_{p/s} & ::= & str_{\text{pre}} \cdot x \cdot str_{\text{suf}} \\
\text{Field count range} & FC & ::= & [n, (n \mid \infty)] \\
\text{Symbolic variable} & x & & \\
\text{Natural number} & n & &
\end{array}
$$

Fig. 9. Symbolic string abstract domain. SaSh models expansion results with the symbolic strings abstract domain, which are either concrete strings or symbolic variables carrying concrete prefix/suffixes and a field-count range.

Parameter expansion: Parameter expansion replaces variable references with their values, or other results under one of POSIX's eight different alternative value or five different string manipulation transformations. In the general case, SaSh supports unmodified parameter expansion and all parameter formats. For parameter expansion, SaSh looks up whether there is a symbolic string mapped to the variable in its environments. If not, it binds the variable to a fresh unconstrained symbolic string (ghost binding) so that future accesses can reference it—capturing the POSIX semantics that unbound variables may be defined by the environment, or if not, become the empty string.

Parameter formats (*e.g.*, `${X-dflt}`, `${X:+rplc}`) introduce more complexity atop this basic template, because they express conditional logic. SaSh deals with alternative-value formats by forking its analysis state if a variable appearing in a format expansion is symbolic: one case constrains the variable to be non-empty, while in the other case the result is the default value with a constraint that the variable is empty. SaSh similarly handles the other alternative-value formats.

SaSh supports string-manipulating parameter expansion transformations (*e.g.*, `${X%suf}`, `${X#pre}`) with a general case and a special case. In the general case, if a variable is bound in SaSh's environment to a non-constant, SaSh expands these parameters with a fresh unconstrained symbolic variable. In the case that the variable has a statically-determined constant value, however, SaSh directly implements the string transformation to produce a corresponding constant result.

Field splitting: Field splitting is the process by which expansion splits up strings delimited by any of the `$IFS` characters—typically space, tab, and newline—into distinct strings. After symbolically expanding words into a sequence interleaving concrete strings and symbolic variables, SaSh performs field splitting to obtain a final sequence of symbolic strings. SaSh first traverses the sequence, splitting it at field boundaries according to the `$IFS` characters. Each field may be one of: only constant strings, a single symbolic variable surrounded by constant strings, or multiple symbolic variables among constant strings. SaSh handles each case separately. For fields with only constant strings, SaSh directly merges the strings to obtain one constant string. For

Table 3. Ground-truth evaluation summary and results. Each row first lists information about each program: a name reference, its source, a brief description of the documented bug(s) it contains; the program's size in lines-of-code; and the maximum path depth at which a bug in the program manifests (↓). The right side of the table lists SaSh's results on the programs: the number of bugs detected over the total number of bugs (**D/#B**) and the number of spurious warnings (false positives) SaSh reports for the fixed program (**FP**); the minimum time budget SaSh needs to fully analyze the program in seconds (t); and which features SaSh uses to detect the bug(s) ($\mathcal{F}$) among word expansion (E), command specifications (S), filesystem model (F), and risk-directed exploration (R).

Script name	Source	Bug description	LoC	↓	D/#B | FP	t	$\mathcal{F}$
DigitalOcean snapshot	[129]	Constant `while`	12	0	1/1 | 0	0.78s	E R
Squid init script	[133]	Unknown `rm` target	139	5	4/4 | 0	>60s	S R
Ubuntu backup manager	[20]	Bad `$?` check	766	10	1/1 | 0	>60s	S R
Node.js version manager	[138]	Deletes `/usr/local/*`	247	8	2/2 | 0	>60s	E S F R
iTunes updater	[135]	Deletes drive	4	1	1/1 | 0	0.13s	E S
Steam updater	[134]	Failed `cd` to `rm /*`	450	7	1/1 | 0	>60s	E S F R
NVIDIA driver installer	[131]	Deletes `/usr`	662	13	1/1 | 0	>60s	S F R
MongoDB backup script	[137]	Typo causes DB loss	15	0	18/18 | 0	0.40s	E S R
Debian debootstrap	[9]	Empty `$2` to `cwd` loss	405	31	1/1 | 0	>60s	E S F R
Debian debootstrap 2	[120]	Deletes supplied path	602	45	0/1 | 0	>60s	E S F
More buggy scripts	App. A		~21.1	~1.1	85/85 | 0	~2.39s	25E 35S 23F 21R
Total		**Arith. mean (~)**	~71.8	~2.9	115/116 | 0	~14.08s	32E 44S 28F 29R

fields with a single symbolic variable, SaSh folds the constant strings before and after the variable into that variable's prefix and suffix. For fields with multiple symbolic variables, SaSh creates a fresh symbolic variable with field-count capturing the full range of possible fields in the concatenation of the original variables, and folds constant strings before the first variable into the prefix and strings after the last variable into the suffix.

In the rare case that `$IFS` is symbolic (`$IFS` is usually set to a constant), SaSh does not attempt field splitting. It issues a warning and approximates the entire expansion with a fresh, unconstrained symbolic variable.

6 Finding More Bugs in Less Time

Even under SaSh's optimistic execution model (*Cf.* §3.2), shell programs can exhibit very large state spaces, growing exponentially with not just each explicit branch and loop, but also during expansion and when interacting with the filesystem. This section describes optimizations that manage this space and find bugs as quickly as possible.

Risk-directed exploration: SaSh's base symbolic execution can spend substantial time in bug-free code before it reaches program fragments most likely to cause serious bugs. To reach these fragments quickly, SaSh performs a targeted exploration phase before regular symbolic execution that focuses on dangerous execution paths. In this phase, SaSh explores a small set of possible executions, prioritizing paths that include dangerous command invocations most likely to be associated with bugs. Specifically, at all conditional branches, SaSh selects only one path to explore (instead of both) by ranking the two branches with a local *risk score*. The risk score is the number of commands appearing literally in the region AST pre-expansion which have at least one invocation specification with a destructive effect.

For example, for the following program, SaSh's risk-directed exploration selects only the **then** branch (due to the presence of `rm`) and ignores the **else** branch entirely.

```
if [ -e $X ]; then rm $Y; else echo skip; fi
```

By extending this approach to decide every conditional branch, this phase guides the search to quickly reach dangerous regions with potentially catastrophic bugs—even when they appear deep inside large programs (*Cf.* §7.3).

State merging and assertion filtering: Exploration can often lead to multiple traces that are equivalent from the analysis's perspective. For example, the following two branches only differ in the contents of `a`, which SaSh does not model:

```
if cmd; then echo "1" > a; else echo "2" > a; fi
```

The forked states resulting from each branch are equivalent, so SaSh merges them to avoid redundant work analyzing the rest of the program. Generally, SaSh only maintains distinct states that diverge in terms of the modeled semantics.

SMT solver calls carry overhead from loading SaSh's encoded symbolic state into the solver before even answering specific queries. In light of this per-call overhead, SaSh avoids invoking the solver on trivial semantic assertions by performing best-effort filtering: during symbolic execution, SaSh inspects each generated assertion and discards it if it can trivially determine that it must be satisfiable—*e.g.*, if the current path condition already entails it or if its concrete values refer only to non-critical paths. For instance, if the path condition includes the constraint that `$v = ""`, SaSh can answer questions about whether `$v` equals any constant without asking the solver.

Time budget allocation: To decide how to best allocate a fixed time budget across multiple analysis stages, SaSh stages execution into risk-directed exploration, full symbolic execution, and constraint solving. The specific distribution is a heuristic informed by the relative costs of these stages.

SaSh partitions a fixed time budget across its analysis stages to prevent any single component from dominating runtime—capping risk-directed exploration at $t/3$, reallocating unused time (up to $t/2$) to symbolic execution, and reserving the remainder for constraint solving. SaSh also supports serializing its symbolic state at the timeout boundary to resume analysis in a subsequent run.

7 Evaluation

We evaluate SaSh on real shell programs along four axes: its completeness at identifying known, well-documented bugs in a controlled setting (§7.1); comparison to the industry-standard ShellCheck linter on the same programs (§7.2); its runtime performance (§7.3); and its ability to find new, previously undiscovered bugs in the wild (§7.4).

Benchmarks: We use the following four benchmark sets:

- Ⓑ 61 real shell programs totaling 116 documented **bugs** and their corresponding fixes, from web forums, StackOverflow, Unix & Linux Stack Exchange, GitHub, and GitLab, including disasters from Steam (§2), iTunes [135], and Nvidia drivers [131] (§7.1–7.3).
- Ⓥ 19 semantically-equivalent but syntactically different **variants** (containing 42 bugs) from a subset of the above programs (§7.1–§7.2).
- Ⓚ 119 programs from the **Koala** benchmark suite [69], providing a diverse and well-studied set of *correct* real-world shell programs (§7.3).
- Ⓡ Several open-source **repositories**, including projects such as PyTorch, Next.js, and the P4 compiler (§7.4).

Hardware & software setup: All experiments were conducted on a single `pc51` Cloudlab node [28]. SaSh runs on Python v3.10.18 and uses the Z3 solver v4.15.4. ShellCheck comparisons use v0.11.0, configured with `--enable all` and `--severity info` to maximize reporting.

7.1 Bug Detection Effectiveness

We apply SaSh to the set of programs with documented bugs (Ⓑ) to judge its bug detection effectiveness against established ground truths. We mark a bug as detected by SaSh when it produces a warning describing the precise undesired behavior, such as a critical path potentially being deleted or a constant conditional.

To judge whether SaSh can distinguish between buggy and correct code (its *discriminative power*), we perform a negative control experiment by applying it to the fixed versions of the programs to see whether it recognizes the absence of a bug, *i.e.*, whether it still warns even though the bug has been fixed (a false positive).

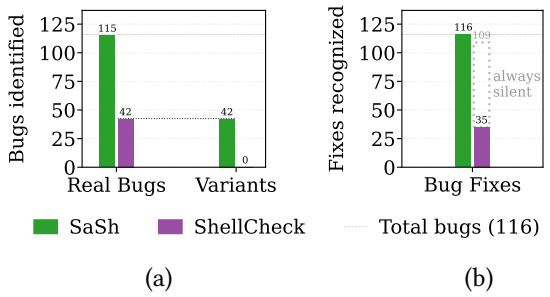


Fig. 10. Effectiveness results (number of bugs detected). Higher is better. Subplot a shows number of bugs detected (*Cf.* §7.2 for variants); SaSh identifies 115 of the 116 bugs in 60 of the 61 programs, and ShellCheck identifies 42 bugs in 19 programs. Subplot b shows number of bug fixes recognized (*i.e.*, warning absent after fixing the bug); SaSh recognizes all fixes; ShellCheck recognizes 35 (in addition to the 74 bugs it never identified in the first place).

Tab. 3 summarizes SaSh's bug-detection effectiveness for a subset of 10 benchmarks (full table in App. A), and Fig. 10a summarizes the results of applying SaSh to every program ("Real Bugs" on the left side). At a high level, SaSh identifies 115/116 bugs across 60/61 programs, including the Steam updater bug (§2). It also identifies 42/42 bugs across the 19 variant programs.

Fig. 10b summarizes the results of applying SaSh to every fixed program in the evaluation set with the left bar counting the number of bug fixes it recognizes—*i.e.*, bug-fix instances for which SaSh no longer warns about the now-fixed bug (higher is better, meaning fewer false positives). SaSh does not warn about any of the fixed bugs in the evaluation set, showing that SaSh does not blindly report on patterns and can distinguish between buggy and fixed code.

Discussion: There are several notable bugs. The SLURM cluster-overwrite bug, in which the program includes a `find` command piped into an `xargs` invocation of `mv`, moves files into the same destination, causing data loss in the scenario where the target is not a directory [128]. SaSh models `xargs` akin to a loop, unrolling it into (three) repeated sequential invocations of its subcommand with unconstrained symbolic arguments. It identifies the repeated move via `mv`'s specification, as well as the potential data loss—`mv`'s precondition asserts that the destination should be read if it is a file (which, in the case of the bug, it is not on the second invocation), reporting a conditional warning: data loss occurs if the target is not a pre-existing directory.

The only instance where SaSh fails to detect a bug is in the second Debian debootstrap script, because the bug depends on developer intent: the program installs a Debian base system in a subdirectory of another already-installed

system [24, 120]; the bug manifests when a user invokes the program with the `--print-debs` flag and a target directory: the script prints a list of dependencies and then deletes the target directory. SaSh could be configured to report this bug by marking positional parameters as critical paths, but generally deleting a user-supplied path is not a bug, thus SaSh by default does not assert against this behavior to avoid a flood of spurious warnings.

7.2 Comparison to ShellCheck

ShellCheck, a syntactic linter that warns about shell-related "code smells" [115], overlaps in intention with SaSh. The first experiment compares it against SaSh on all the programs in Tab. 3 (Ⓑ). As syntactic code smells are too broad relative to SaSh's precise semantic warnings, we consider that ShellCheck identifies a bug if it produces *any* warning that is related to the issue (even if it does not precisely describe it). For example, in the Ubuntu backup manager program, ShellCheck does not flag the constant condition, but issues a generic warning about avoiding `$?`; nonetheless, we consider this as ShellCheck reporting this bug. We also consider that ShellCheck fails to recognize a fix whenever it produces the same warning in the fixed version of the program.

Fig. 10a shows the result of this experiment, comparing ShellCheck's bug-detection effectiveness (purple) against SaSh's (green). In 42 instances, ShellCheck produces some warnings that, if inspected and interpreted correctly, could allow a user to identify the underlying bug. ShellCheck's bug-finding effectiveness is correlated with the presence of certain patterns, such as using `$?` in conditionals, unquoted variables, or invoking `rm` with a variable operand suffixed with a slash. It has also retrofitted some checks specifically after major disasters, including the Steam updater bug (§2) and the Nvidia driver installer bug [131] which it detects by warning about the specific pattern that was involved in each disaster [113]. Almost half of the bugs ShellCheck identifies (18/42) come from the MongoDB backup script, where developers mistakenly use C-style comments (`//`) instead of shell comments (`#`) in their program, causing variable assignments to not propagate.

Fig. 10b plots ShellCheck's fix recognition against SaSh's on the *fixed* versions of the benchmarks. The bottom, darker portion shows the number of fixes for which ShellCheck no longer warns about the original bug in 35 out of the 42 bugs that it identifies in the first place. It still warns about the now-fixed bug in seven instances. The dotted area above the bar indicates the remaining set of 74 bugs for which ShellCheck is always silent: it issued no warning in the first place, and continues not to warn for the fix.

Variants: The second experiment demonstrates the brittleness of ShellCheck's syntactic approach (right half of Fig. 10a) by comparing SaSh and ShellCheck on *variants*: semantically-equivalent but syntactically different programs of all 42 bugs

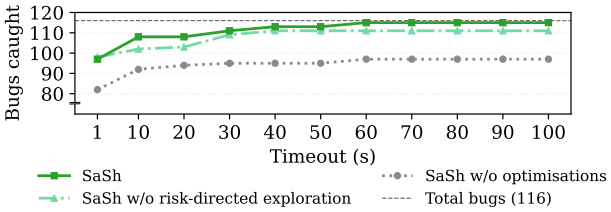


Fig. 11. Number of bugs SaSh detects within time budget. The x axis indicates different time budgets, and the y axis indicates the number of bugs SaSh detects within those time budgets. Results are from three different configurations: the base, non-optimistic, symbolic execution engine, optimistic symbolic execution without risk-directed exploration, and full SaSh.

ShellCheck identifies (Ⓥ). For instance, in the aforementioned MongoDB backup program, assigning `c="//"` and replacing all instances of `//` with `$c` results in a program that is semantically equivalent, manifests the same set of underlying issues, but does not trigger any of ShellCheck's warnings. A variant of the Steam updater program (§2) moves the trailing slash from the `rm` invocation to the variable assignment: `STEAMROOT="$(cd ...)/"`. ShellCheck identifies none of the variant bugs; SaSh identifies the same bug manifestation across all of them.

7.3 Performance Analysis

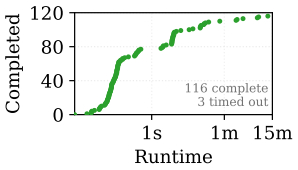
This section reports on SaSh's execution time when analyzing shell programs and the benefits of its optimizations (§6). Because SaSh identifies as many bugs as it can within a time budget, optimizations of its run time also affect its bug-finding ability for a given budget.

Bug-finding across time budgets: Fig. 11 reports on SaSh's bug-finding performance under different time budgets, and with different optimizations enabled on the set of documented buggy programs (Ⓑ). The three configurations compared are (1) SaSh's base, non-optimistic, symbolic execution engine without any of the optimizations described in §6, (2) optimistic symbolic execution without risk-directed exploration, and (3) full SaSh. SaSh without any optimizations detects 82 bugs within one second, and up to 97 bugs, saturating at 60s. Without risk-directed exploration, SaSh detects 98 bugs within one second, and up to 111 bugs, saturating at 40s. Full SaSh detects 97 bugs within one second, and up to 115 bugs, saturating at 60s.

Tab. 3 summarizes the runtime performance of SaSh for every documented buggy program in the evaluation (Ⓑ), under a timeout of 60s. Excluding programs for which SaSh times out, the average runtime per program is 0.75s, with a minimum of 0.02s and a maximum of 23.31s. Although SaSh hits its time budget (and hence terminates without fully completing analysis) on 11 of the benchmarks, it nonetheless analyzes enough of all programs to identify the bugs present in all but one of them, as discussed in §7.1.

The largest and most complex programs of the evaluation set enjoy the benefits of SaSh's risk-directed exploration and other optimizations. A notable example is the first Debian debootstrap script, where an empty `$2` results in `PWD` being deleted, which can be the user's home directory (as happened to the bug reporter) [9]. This script is 518 lines long, and the dangerous `rm -rf` invocation appears at line 510, with 276 lines separating the invocation with the earlier, offending line that assigns `rm`'s later target to `PWD`. In-between, among other logic, is an argument parsing routine implemented as a 20-branch **`case`** statement surrounded by a **`while`** loop. This bug highlights the value of SaSh's targeted execution passes, as naïve exploration inevitably gets stuck exploring paths in the argument parsing logic, exhausting SaSh's time budget without exploring other parts of the program. With its targeted execution, SaSh goes from unable to identify the bug within one *hour*, to identifying it within 30s. SaSh's optimizations enable identifying bugs hidden deep within the logic of large programs (*e.g.*, the Steam updater, the Debian debootstrap program, and the Ubuntu backup manager).

Time for complete analysis: To judge the time budget SaSh requires to *completely* analyze programs, we report on the time SaSh requires to fully analyze all scripts in the Koala benchmark suite [69] (Ⓚ). The suite consists of 119 correct, diverse scripts with a wide variety of characteristics. The plot below depicts the results: SaSh can completely analyze 77 (65%) of the programs within one second, 110 (92%) within one minute, and all but three within 15 minutes. The last three do not finish even after 1 hour due to the large number of loops, conditionals, and filesystem updates that result in extensive solver overhead. Indeed, SaSh expects to not be able to fully analyze all programs—its optimizations (§6) help surface bugs even under such conditions.



7.4 Identifying New Bugs in the Wild

To demonstrate SaSh's effectiveness at identifying new bugs, we applied it to several open-source repositories in the wild in the weeks before submission (Ⓡ). SaSh has already identified 70 unique bugs across 44 open-source projects—including PyTorch, Next.js, and Kubernetes—each of which we manually inspected, validated, and reported to the maintainers alongside fixes. At the time of writing, 27 fixes have already been accepted. The table below summarizes a selection of notable bugs SaSh identified; it lists the project, the script's domain, and the number of occurrences of the reported bug (**#B**).

Project	Domain	#B
PyTorch [102]	CI	1
Kubernetes [66]	CI	3
Next.js [141]	CI	1
P4 Compiler [7]	Build	15
vLLM [142]	CI	4
AFFiNE [122]	Updater	3
Moby [75]	CI	2
Total (*Cf.* App. B)	...	70

The issues identified span dangerous path deletions that could result in critical data loss (P4 Compiler), improper exit-code handling that results in program crashes or dead code (AFFiNE, vLLM), expansion mistakes causing bad control (Kubernetes), and unbound references that result in core functionality never executing (PyTorch, Kubernetes, Next.js, Moby). Beyond the selection summarized here, the remaining bug reports include more dangerous side effects, improper handling of user inputs, and unexpected failure scenarios; App. B describes the full set of discovered bugs.

8 Related Work

Research on shell correctness: Previous work studies shell usage in the wild [26, 109], investigating the prevalence and severity of shell bugs such as the ones Sash targets. Prior work also develops formal semantics for shell languages [58, 64] and type systems [116] for interprocess communication in fully expanded shell pipelines. SaSh focuses on high-performance, automated, ahead-of-time analysis of real shell programs like the ones found in the wild—supporting broader effects, expansion reasoning, and composition features far beyond pipeline composition.

Filesystem modeling and effect analysis: A body of research formalizes the behavior of POSIX-style filesystems, including path-level semantics [117], local updates with global consequences [81], and concurrency [6]. In contrast to all that work, SaSh supports automated analysis on large and complex shell programs—including expansion, and filesystem effects through unknown variables and paths.

Rehearsal [112, 145] supports a filesystem model implemented as a flat map between paths and path states. Unlike Rehearsal, SaSh can reason about commands that can have arbitrary side-effects over unknown paths.

Analysis of orchestration languages: Prior work has focused on analyses of orchestration languages [74, 112, 145], including configuration verification [112], security-oriented taint analysis [74], and example-driven configuration repair for commands [21] or system configurations [145]. SaSh, instead, focuses on shell programs, supports the full set of POSIX expansions, can reason about arbitrary inputs, and models complex filesystem effects.

ABash [74] develops arity reasoning for command arguments with some support for expansion. Contrary to ABash, SaSh supports all common shell constructs such as conditionals, loops, and pipes, models command conditions, and reasons about filesystem effects.

Symbolic and concrete execution: Symbolic execution is an established technique for program analysis [2, 10, 11],

with scalability improvements stemming from state merging and path selection [13, 67], constraint solving [23], and summary-based reasoning [1]. SaSh advances these symbolic execution techniques by incorporating domain-specific insights specific to the shell.

KLEE [10] combines symbolic and concrete execution of LLVM bytecode along with a system-call-level environment modeling. SaSh targets the POSIX semantics of shell programs, including composition and expansion, performs symbolic-only execution, and allows commands to be written in any language.

9 Conclusion

SaSh shows that ahead-of-time analysis of shell programs can be implemented practically despite the shell's dynamism and environment interactions—successfully identifying catastrophic bugs before execution. SaSh identifies all but one of the 116 unwanted behaviors in a set of real shell programs and has already uncovered previously unknown bugs in widely used open-source systems.

Acknowledgments

We would like to thank the Brown CS2952R (Fall'25) participants, Akshay Narayan, and Deepti Raghavan for their input on several iterations of this paper; and Felix Stutz, Georgios Liargkovas, and Anirudh Narsipur for early explorations towards ahead-of-time correctness guarantees in the context of the Shell; This material is based upon research supported by NSF awards CNS-2247687, CNS-2312346, and CCF-2340479; NSF GRFP grant no. 2439559; DARPA contract no. HR001124C0486, an Amazon Research Award (Fall 2024), a Google ML-and-Systems Junior Faculty Award, a seed grant from Brown University's Data Science Institute, and a BrownCS Faculty Innovation Award.

References

[1] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS/ETAPS '08, pages 367–381, Berlin, Heidelberg, 2008. Springer.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018.

[3] Base. Base node. https://github.com/base/node. Accessed: 2026-03-16.

[4] batocera linux. batocera.linux. https://github.com/batocera-linux/batocera.linux. Accessed: 2026-03-29.

[5] BeyondDimension. Steamtools. https://github.com/BeyondDimension/SteamTools, n.d. Accessed: 2026-04-01.

[6] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 83–98, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[8] Pádraig Brady. File replacement on unix. https://www.pixelbeat.org/docs/unix_file_replacement.html. Accessed: 2026-03-10.

[9] Debian bug report. Debian bug report logs: –print-debs removes current working dir. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=337230, 2005. Accessed: 2026-02-14.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '08, pages 209–224, USA, December 2008. USENIX Association.

[11] Cristian Cadar and Dawson Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In Patrice Godefroid, editor, *Model Checking Software*, pages 2–23, Berlin, Heidelberg, 2005. Springer.

[12] ch32 rs. ch32-data. https://github.com/ch32-rs/ch32-data, n.d. Accessed: 2026-04-01.

[13] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, February 2012.

[14] ClouGence. hasor. https://github.com/ClouGence/hasor, n.d. Accessed: 2026-04-01.

[15] comma.ai. Openpilot. https://github.com/commaai/openpilot. Accessed: 2026-03-16.

[16] Unix Linux Community. Move file in current date folder through shell script - shell programming and scripting - unix linux community. https://community.unix.com/t/move-file-in-current-date-folder-through-shell-script/384145. Accessed: 2026-03-10.

[17] Valve Corporation. Steam. https://store.steampowered.com, 2025. Accessed: 2025-10-10.

[18] Anthony Cozzie, Murph Finnicum, and Samuel T. King. Macho: Programming with man pages. In *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, Napa, CA, May 2011. USENIX Association.

[19] Crawl4AI. Crawl4ai. https://github.com/unclecode/crawl4ai. Accessed: 2026-03-16.

[20] Maximiliano Curia. Ubuntu backup-manager; commit 59cdd32. https://git.launchpad.net/ubuntu/+source/backup-manager/commit/lib/backup-methods.sh?id=59cdd322f7772d40b6aff25b6720796271f4deba, 2016. Accessed: 2025-10-10.

[21] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. Nofaq: synthesizing command repairs from examples. In *2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '17, pages 582–592, New York, NY, USA, 2017. Association for Computing Machinery.

[22] day0ops. gloo-gateway-2.1-demo. https://github.com/day0ops/gloo-gateway-2.1-demo. Accessed: 2026-03-27.

[23] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[24] Debian. Debootstrap. https://wiki.debian.org/Debootstrap. Accessed: 2026-02-14.

[25] Yann Dirson and Julian Gilbey. checkbashisms(1) - linux man page. https://manpages.debian.org/trixie/devscripts/checkbashisms.1.en.html, 1998. Accessed: 2026-03-30.

[26] Yiwen Dong, Zheyang Li, Yongqiang Tian, Chengnian Sun, Michael W. Godfrey, and Meiyappan Nagappan. Bash in the wild: Language usage, code smells, and bugs. *ACM Trans. Softw. Eng. Methodol.*, 32(1), February 2023.

[27] dreysanox. Toolsave. https://github.com/dreysanox/ToolSave. Accessed: 2026-03-27.

[28] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. In *2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 1–14, USA, 2019. USENIX Association.

[29] edeliver. edeliver. https://github.com/edeliver/edeliver, n.d. Accessed: 2026-04-01.

[30] EhsanAramide. V2m. https://github.com/EhsanAramide/V2M. Accessed: 2026-03-27.

[31] erikfrey. bashreduce. https://github.com/erikfrey/bashreduce. Accessed: 2026-03-27.

[32] ErLinErYi. Plantsvszombies. https://github.com/ErLinErYi/PlantsVsZombies, n.d. Accessed: 2026-04-01.

[33] Vidar Holen et al. Shellcheck: A shell script static analysis tool. https://www.shellcheck.net/, 2012. Accessed: 2025-10-10.

[34] Unix & Linux Stack Exchange. Bash: move files of specific pattern. https://unix.stackexchange.com/questions/231513/bash-move-files-of-specific-pattern. Accessed: 2026-03-10.

[35] Unix & Linux Stack Exchange. How to recover files i deleted now by running rm *? [duplicate]. https://unix.stackexchange.com/questions/101237/how-to-recover-files-i-deleted-now-by-running-rm. Accessed: 2026-03-10.

[36] Server Fault. Monday morning mistake: sudo rm -rf –no-preserve-root /. https://serverfault.com/questions/587102/monday-morning-mistake-sudo-rm-rf-no-preserve-root. Accessed: 2026-03-10.

[37] Server Fault. why doesn't mkdir -p /tmp/one/two/tree print anything when directories exist? https://serverfault.com/questions/702207/why-doesnt-mkdir-p-tmp-one-two-tree-print-anything-when-directories-exist. Accessed: 2026-03-10.

[38] Fred78290. caker. https://github.com/Fred78290/caker. Accessed: 2026-03-27.

[39] Kyle Fuller. swiftenv. https://github.com/kylef/swiftenv, n.d. Accessed: 2026-04-01.

[40] gabrie30. ghorg. https://github.com/gabrie30/ghorg. Accessed: 2026-03-29.

[41] gihrig. dotfiles-ooloth. https://github.com/gihrig/dotfiles-ooloth. Accessed: 2026-03-29.

[42] GitHub. Agent blindly mkdir -p then destructively rmdir on rollback without checking existing contents · issue #24787. https://github.com/anthropics/claude-code/issues/24787. Accessed: 2026-03-10.

[43] GitHub. Bash tool: newline characters in command string treated as spaces, causing rm -rf to delete unintended directories · issue #24319. https://github.com/anthropics/claude-code/issues/24319. Accessed: 2026-03-10.

[44] GitHub. [bug fix]: src/download_all_github_repos.sh. https://github.com/djeada/Bash-Scripts/commit/3e28d0238393215d5b70432d067b719359176143. Accessed: 2026-02-19.

[45] GitHub. [bug] unsafe rm command execution deletes entire home directory · issue #12637. https://github.com/anthropics/claude-code/issues/12637. Accessed: 2026-03-10.

[46] GitHub. [critical] bash tool deletes entire project root after command interrupt - pwd state corruption · issue #17410. https://github.com/anthropics/claude-code/issues/17410. Accessed: 2026-03-10.

[47] GitHub. fix: add .env.example and fix deploy script cp command. https://github.com/ClaudioPaulo71/tib-cigar/commit/88e009fc6be6c51f298ccf28e2642060485a5b9a. Accessed: 2026-03-10.

[48] GitHub. fix bug for check zsh (#6798). https://github.com/ohmyzsh/ohmyzsh/commit/8f0ff4bb63a8fd26741128a851c224af323eb772. Accessed: 2026-02-19.

[49] GitHub. Fix inexistent folder on actualbudget update script (#1444). https://github.com/community-scripts/ProxmoxVE/commit/695802152856a8bd011ad060a70082dd1cf91ead. Accessed: 2026-02-19.

[50] GitHub. Fix symlink check bug in dotfiles backup. https://github.com/spf13/spf13-vim/commit/52355abace2a055ba41cf905f9e7f78d14f33398. Accessed: 2026-02-19.

[51] GitHub. Fixed bug introduced when fixing issue 896. https://github.com/ohmyzsh/ohmyzsh/commit/5a5c93b33493b47d34dd470eb851aaf24fa87536. Accessed: 2026-02-19.

[52] GitHub. Fixing a bug in install_as_script. https://github.com/nvm-sh/nvm/commit/5904d41b253e03d7ad0f3b58ce081e46e44f3c9a. Accessed: 2026-02-19.

[53] GitHub. Fixing install_home.sh - failing grep. https://github.com/babun/babun/commit/4e49aebca73ff7d6ecb587a417945d21c4974a3f. Accessed: 2026-02-19.

[54] GitHub. fix(msi): launch installer [skip ci]. https://github.com/VSCodium/vscodium/commit/97eb57c196159bb0bcebc04adf5d5dac0885fb4d. Accessed: 2026-02-19.

[55] GitHub. opam deleted home directory (issue #3316). https://github.com/ocaml/opam/issues/3316. Accessed: 2026-02-19.

[56] GitHub. Temporary files not cleaned up on windows (tmpclaude-*-cwd, nul) · issue #17925. https://github.com/anthropics/claude-code/issues/17925. Accessed: 2026-03-10.

[57] GitHub. Octoverse 2025. https://octoverse.github.com/, 2025. Accessed: 2026-04-01.

[58] Michael Greenberg and Austin J. Blatt. Executable formal semantics for the posix shell. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[59] The Open Group. Consequences of shell errors. https://pubs.opengroup.org/onlinepubs/9799919799.2024edition/utilities/V3_chap02.html#tag_19_08_01, 2024. Accessed: 2026-03-16.

[60] The Open Group. The open group base specifications issue 8, ieee std 1003.1-2024 edition. https://pubs.opengroup.org/onlinepubs/9799919799.2024edition/, 2024. Accessed: 2026-02-19.

[61] The Open Group. Special built-in utilities. https://pubs.opengroup.org/onlinepubs/9799919799.2024edition/utilities/V3_chap02.html#tag_19_15, 2024. Accessed: 2026-02-19.

[62] huangrt01. Cs-notes. https://github.com/huangrt01/CS-Notes, n.d. Accessed: 2026-04-01.

[63] ipinfo. cli. https://github.com/ipinfo/cli, n.d. Accessed: 2026-04-01.

[64] Nicolas Jeannerod, Claude Marché, and Ralf Treinen. A Formally Verified Interpreter for a Shell-like Programming Language. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments*, volume 10712, Heidelberg, Germany, July 2017.

[65] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.*, 35(2), January 2026.

[66] Kubernetes. test-infra. https://github.com/kubernetes/test-infra. Accessed: 2026-03-29.

[67] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 193–204, New York, NY, USA, June 2012. Association for Computing Machinery.

[68] Evangelos Lamprou, Seong-Heon Jung, Mayank Keoliya, Lukas Lazarek, Konstantinos Kallas, Michael Greenberg, and Nikos Vasilakis. Caruca: Effective and efficient specification mining for opaque software components. *arXiv preprint arXiv:2510.14279*, 2025.

[69] Evangelos Lamprou, Ethan Williams, Georgios Kaoukis, Zhuoxuan Zhang, Michael Greenberg, Konstantinos Kallas, Lukas Lazarek, and Nikos Vasilakis. The Koala benchmarks for the shell: Characterization and implications. In *2025 USENIX Annual Technical Conference*, USENIX ATC '25, Santa Clara, CA, 2025. USENIX Association.

[70] LibreELEC. Libreelec.tv. https://github.com/LibreELEC/LibreELEC.tv, n.d. Accessed: 2026-04-01.

[71] lintian team. Lintian. https://wiki.debian.org/Lintian, 1998. Accessed: 2025-10-25.

[72] LinuxQuestions.org. [solved] accidently deleted my home directory. https://www.linuxquestions.org/questions/linux-newbie-8/accidently-deleted-my-home-directory-4175715793/. Accessed: 2026-03-10.

[73] LinuxQuestions.org. [solved] mv is not working right. https://www.linuxquestions.org/questions/linux-general-1/mv-is-not-working-right-782971/. Accessed: 2026-03-10.

[74] Karl Mazurak and Steve Zdancewic. Abash: finding bugs in bash scripts. In *2007 Workshop on Programming Languages and Analysis for Security*, PLAS '07, pages 105–114, New York, NY, USA, 2007. Association for Computing Machinery.

[75] Moby. Moby: An open-source project for the container ecosystem. https://github.com/moby/moby. Accessed: 2026-02-14.

[76] Mondego. Sourcerercc. https://github.com/Mondego/SourcererCC, n.d. Accessed: 2026-04-01.

[77] multigres. multigres. https://github.com/multigres/multigres, n.d. Accessed: 2026-04-01.

[78] netdata. netdata. https://github.com/netdata/netdata, n.d. Accessed: 2026-04-01.

[79] Akash Network. cosmos-omnibus. https://github.com/akash-network/cosmos-omnibus. Accessed: 2026-03-27.

[80] Nick Nisi. dotfiles. https://github.com/nicknisi/dotfiles. Accessed: 2026-03-29.

[81] Gian Ntzik and Philippa Gardner. Reasoning about the posix file system: local update and global pathnames. In *2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '15, pages 201–220, New York, NY, USA, 2015. Association for Computing Machinery.

[82] OpenDev. Fix unbound variable error in scripts/collect-test-info.sh. https://opendev.org/openstack/bifrost/commit/7fb32dd599c0d8b5d231cd30c5a8a1ef5142110a. Accessed: 2026-02-19.

[83] OpenSC. Opensc. https://github.com/OpenSC/OpenSC. Accessed: 2026-03-29.

[84] Stack Overflow. Bash variable defaulting doesn't work if followed by pipe (bash bug?). https://stackoverflow.com/questions/26526776/bash-variable-defaulting-doesnt-work-if-followed-by-pipe-bash-bug. Accessed: 2026-02-19.

[85] Stack Overflow. Check if files of a given type exist in bash shell. https://stackoverflow.com/questions/48854121/check-if-files-of-a-given-type-exist-in-bash-shell. Accessed: 2026-03-10.

[86] Stack Overflow. Customized function for error logs scripting. https://stackoverflow.com/questions/49043790/customized-function-for-error-logs-scripting. Accessed: 2026-02-19.

[87] Stack Overflow. error with bash script using rsync to copy directory with space in directory name. https://stackoverflow.com/questions/48706718/error-with-bash-script-using-rsync-to-copy-directory-with-space-in-directory-nam. Accessed: 2026-02-19.

[88] Stack Overflow. file copying in while loop is executing for one time only. https://stackoverflow.com/questions/49605847/file-copying-in-while-loop-is-executing-for-one-time-only. Accessed: 2026-02-19.

[89] Stack Overflow. How can i stop my script to overwrite existing files. https://stackoverflow.com/questions/65666474/how-can-i-stop-my-script-to-overwrite-existing-files. Accessed: 2026-02-19.

[90] Stack Overflow. I am having a hard time why the if statements don't work in shellscript. https://stackoverflow.com/questions/48919816/i-am-having-a-hard-time-why-the-if-statements-dont-work-in-shellscript. Accessed: 2026-02-19.

[91] Stack Overflow. Issue in mkdir output to variable. https://stackoverflow.com/questions/73501167/issue-in-mkdir-output-to-variable. Accessed: 2026-02-19.

[92] Stack Overflow. mkdir not working in centos 7? https://stackoverflow.com/questions/48674985/mkdir-not-working-in-centos-7. Accessed: 2026-02-19.

[93] Stack Overflow. Not able to overwrite the file properly through unix script. https://stackoverflow.com/questions/49138117/not-able-to-overwrite-the-file-properly-through-unix-script. Accessed: 2026-02-19.

[94] Stack Overflow. restoring deleted file using bash in unix. https://stackoverflow.com/questions/42844610/restoring-deleted-file-using-bash-in-unix. Accessed: 2026-02-19.

[95] Stack Overflow. Trying to iterate through files stored in variables. https://stackoverflow.com/questions/49562688/trying-to-iterate-through-files-stored-in-variables. Accessed: 2026-02-19.

[96] Stack Overflow. Why find piped to xargs mv deleted my files? https://stackoverflow.com/questions/64098546/why-find-piped-to-xargs-mv-deleted-my-files. Accessed: 2026-03-10.

[97] Stack Overflow. Stack overflow 2025 developer survey. https://survey.stackoverflow.co/2025/technology, 2025. Accessed: 2026-03-29.

[98] Steve Parker. rm -rf / …a confession. https://www.shellscript.sh/examples/rm-rf/, 2016. Accessed: 2025-10-10.

[99] petervislocky. Theme-switcher. https://github.com/petervislocky/Theme-Switcher. Accessed: 2026-03-27.

[100] PgBouncer. pgbouncer. https://github.com/pgbouncer/pgbouncer, n.d. Accessed: 2026-04-01.

[101] pluja. whishper. https://github.com/pluja/whishper, n.d. Accessed: 2026-04-01.

[102] PyTorch. Pytorch. https://github.com/pytorch/pytorch. Accessed: 2026-03-16.

[103] Reddit. Claude cli deleted my entire home directory! wiped my whole mac. https://www.reddit.com/r/ClaudeAI/comments/1pgxckk/claude_cli_deleted_my_entire_home_directory_wiped/. Accessed: 2026-03-10.

[104] Reddit. How do i use mv and cp commands within a bash shell script? https://www.reddit.com/r/linuxquestions/comments/1dldolx/how_do_i_use_mv_and_cp_commands_within_a_bash/. Accessed: 2026-03-10.

[105] Reddit. i fell for a troll and deleted my /home directory. https://www.reddit.com/r/linuxquestions/comments/161v8aj/i_fell_for_a_troll_and_deleted_my_home_directory/. Accessed: 2026-03-10.

[106] RenderKit. embree. https://github.com/RenderKit/embree, n.d. Accessed: 2026-04-01.

[107] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

[108] Keefer Rourke. la-capitaine-icon-theme. https://github.com/keeferrourke/la-capitaine-icon-theme, n.d. Accessed: 2026-04-01.

[109] Schröder, Michael and Cito, Jürgen. An empirical investigation of command-line customization. *Empirical Software Engineering*, 27(2):30, 2021.

[110] SegmentFault. Error after modifying mac bash_profile. https://segmentfault.com/q/1010000000158149. Accessed: 2026-03-10.

[111] Serverless. Serverless framework. https://github.com/serverless/serverless. Accessed: 2026-03-27.

[112] Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: a configuration verification tool for puppet. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 416–430, New York, NY, USA, 2016. Association for Computing Machinery.

[113] ShellCheck. find / with rm -rf does produce warning. https://github.com/koalaman/shellcheck/issues/2328. Accessed: 2026-02-24.

[114] ShellCheck. rm -rf warning is disabled by adding parameters to rm, rather than shellcheck. https://www.shellcheck.net/wiki/Issue_910. Accessed: 2026-02-24.

[115] ShellCheck. Shellcheck wiki sitemap. https://www.shellcheck.net/wiki/. Accessed: 2026-02-24.

[116] Michael Sippel and Horst Schirmeier. Process Composition with Typed Unix Pipes. In *12th Workshop on Programming Languages and Operating Systems*, PLOS '23, pages 34–40, New York, NY, USA, 2023. Association for Computing Machinery.

[117] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. A logic of file systems. In *4th USENIX Conference on File and Storage Technologies*, FAST '05, San Francisco, CA, 2005. USENIX Association.

[118] SliTaz. tazpkg. https://github.com/SliTaz-official/tazpkg. Accessed: 2026-03-27.

[119] solid lines. rapidpro-docker. https://github.com/solid-lines/rapidpro-docker. Accessed: 2026-03-27.

[120] Debian Bug Tracking System. Debian bug report logs - #833525 debootstrap: –print-debs removes current working dir. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=833525. Accessed: 2026-02-19.

[121] Tencent. Facedetection-dsfd. https://github.com/Tencent/FaceDetection-DSFD, n.d. Accessed: 2026-04-01.

[122] toeverything. Affine. https://github.com/toeverything/AFFiNE. Accessed: 2026-03-29.

[123] Ask Ubuntu. Accidentally deleted my user's home folder. https://askubuntu.com/questions/1141336/accidentally-deleted-my-users-home-folder. Accessed: 2026-03-10.

[124] Ask Ubuntu. Bash scripting: my script deletes a working folder prematurely. how do i fix? https://askubuntu.com/questions/1356169/bash-scripting-my-script-deletes-a-working-folder-prematurely-how-do-i-fix. Accessed: 2026-02-19.

[125] uplex source repository. Misc: macbuild.sh: fix unbound variable error. https://code.uplex.de/stefan/audiowmark/commit/95db9c5f0b7788aff65f2d85eaedd0c01ba960ac. Accessed: 2026-02-19.

[126] Super User. Help. i ran "find / -mtime +1 -exec rm {} \;" by accident! https://superuser.com/questions/307057/help-i-ran-find-mtime-1-exec-rm-by-accident. Accessed: 2026-03-10.

[127] Super User. mv command lost all files. can find files by locate function but not in file manager. https://superuser.com/questions/998089/mv-command-lost-all-files-can-find-files-by-locate-function-but-not-in-file-man. Accessed: 2026-03-10.

[128] StackOverflow user `Bogdan`. Recovering the files that were deleted in linux by using mv. https://stackoverflow.com/questions/70017719/recovering-the-files-that-were-deleted-in-linux-by-using-mv, 2022. Accessed: 2026-02-14.

[129] Stack Exchange user `cinemafunk`. While loop deletes all files and becomes stuck in loop. https://unix.stackexchange.com/questions/560038/while-loop-deletes-all-files-and-becomes-stuck-in-loop, 2020. Accessed: 2025-10-10.

[130] Stack Overflow user `dhboots`. sh script to replace text in multiple files. https://stackoverflow.com/questions/48195715/sh-script-to-replace-text-in-multiple-files, 2018. Accessed: 2025-10-10.

[131] GitHub user `ginoputrino`. install script does rm -rf /usr for ubuntu #123. https://github.com/MrMEEE/bumblebee-Old-and-abbandoned/issues/123, 2011. Accessed: 2025-10-10.

[132] GitHub user `HairyFotr`. Shellcheck does not detect the steam 'rm -rf /*' bug. https://github.com/koalaman/shellcheck/issues/23#issuecomment-71086960, 2015. Accessed: 2025-10-17.

[133] Red Hat Bugzilla user `jlehtone`. Bug 1102343 - service squid restart sometimes leaves duplicate processes. https://bugzilla.redhat.com/show_bug.cgi?id=1102343, 2014. Accessed: 2025-10-10.

[134] GitHub user `keyvin`. Moved ~/.local/share/steam. ran steam. it deleted everything on system owned by user. #3671. https://github.com/ValveSoftware/steam-for-linux/issues/3671, 2015. Accessed: 2025-10-10.

[135] Slashdot user `michael`. itunes 2.0 installer deletes hard drives. https://apple.slashdot.org/story/01/11/04/0412209/itunes-20-installer-deletes-hard-drives, 2001. Accessed: 2025-10-10.

[136] GitHub user `romkatv`. update: fix bug in upgrade.sh: s/normal/reset/. https://github.com/ohmyzsh/ohmyzsh/commit/f7bf566555a2c0e87deba5dfb3e344f23f4a51bb, 2020. Accessed: 2025-10-10.

[137] Stack Overflow user `SoftTimur`. Deleted database accidentally by a bash script - rescue please. https://stackoverflow.com/questions/55323391/deleted-database-accidentally-by-a-bash-script-rescue-please, 2019. Accessed: 2025-10-10.

[138] GitHub user `tj`. n; commit d36c2b8. https://github.com/tj/n/commit/d36c2b88047e3eae27e80938a13b1313df0c1f82, 2012. Accessed: 2025-10-10.

[139] Priyan Vaithilingam and Philip J. Guo. Bespoke: Interactively synthesizing custom guis from command-line applications by demonstration. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, page 563–576, New York, NY, USA, 2019. Association for Computing Machinery.

[140] Ventoy. Ventoy. https://github.com/ventoy/Ventoy. Accessed: 2026-03-27.

[141] Vercel. Next.js: The react framework. https://nextjs.org/. Accessed: 2026-02-14.

[142] vLLM Project. vllm. https://github.com/vllm-project/vllm. Accessed: 2026-03-16.

[143] wanikua. danghuangshang. https://github.com/wanikua/danghuangshang. Accessed: 2026-03-29.

[144] warp.dev. The state of the cli 2023 edition. https://www.warp.dev/state-of-the-cli-2023#the-gap-between-cli-usage-and-proficiency, 2023. Accessed: 2025-12-02.

[145] Aaron Weiss, Arjun Guha, and Yuriy Brun. Tortoise: Interactive System Configuration Repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, pages 625–636, October 2017.

A Full Program Set

Table 4. Ground-truth evaluation summary and results. Each row first lists information about each program: a short identifier, its source, and name for reference; a brief description of the documented bug(s) the program contains; the program's size in lines-of-code; and the maximum path depth at which a bug in the program manifests (↓). The right side of the table lists SaSh's results on the programs: the number of bugs detected over the total number of bugs (**D/#B**) and the number of false positives SaSh reports for the fixed program (**FP**); the minimum time budget SaSh needs to fully analyze the program in seconds (t); and which features SaSh uses to detect the bug(s) ($\mathcal{F}$) among word expansion (E), command specifications (S), filesystem model (F), and risk-directed exploration (R).

Script name	Source	Bug description	LoC	↓	D/#B | FP	t	$\mathcal{F}$
AutoTest config rename	[130]	Run-once `for` loop	11	0	1/1 | 0	0.15s	E
Log redirection helper	[86]	Redirect to function	27	0	2/2 | 0	0.24s	S
OhMyZsh update script	[136]	Unset variable in `echo`	61	3	2/2 | 0	1.28s	E
DigitalOcean snapshot	[129]	Constant `while`	12	0	1/1 | 0	0.78s	E R
Claude null-output redirect	[56]	Typo overwrites file	2	0	1/1 | 0	0.04s	F
Media file conversion	[127]	Deletes files before `mv`	1	0	2/2 | 0	0.16s	S F R
Move home directory	[72]	Moves user's `$HOME`	2	0	1/1 | 0	0.03s	S F
Obfuscated `rm /`	[105]	Malware deletes `/home`	2	0	2/2 | 0	0.06s	S F
Claude home wipe	[45]	Claude deletes `$HOME`	4	0	2/2 | 0	0.09s	E S R
Filesystem preparation	[92]	Always empty `mkdir` arg	20	4	2/2 | 0	3.30s	E S
Claude cleanup output	[103]	Agent wipes `$HOME`	2	0	1/1 | 0	0.11s	E S
Unset PATH startup	[110]	Unset `PATH`	14	0	12/12 | 0	0.21s	S
Backup wipe	[96]	Moves files to missing dir	1	0	1/1 | 0	0.15s	S F R
Accidental recursive delete	[35]	Wildcard `rm` deletes cwd	3	0	3/3 | 0	0.12s	E S F R
Deleted home directory	[123]	Deletes `/home/user`	1	0	1/1 | 0	0.06s	S F
Claude temp-file cleanup	[42]	Deletes only file copy	4	1	2/2 | 0	0.12s	S F R
Domain folder creator	[91]	Captures `mkdir` output	8	1	1/1 | 0	0.12s	S
System restoration script	[94]	Empty `$1` to stuck program	15	0	2/2 | 0	0.43s	S
No-preserve-root delete	[36]	Deletes `/` via extra `rm` arg	1	0	1/1 | 0	0.08s	S F
Broken multi-file rename	[104]	Passes multiple sources to `mv`	5	0	4/4 | 0	0.17s	S F R
Glob test mismatch	[85]	Quoted glob in file check	2	2	1/1 | 0	0.07s	E S R
Directory clear accident	[126]	Deletes system files	1	0	2/2 | 0	0.11s	S F
AIX server data gather	[98]	Failed `mktemp` to data loss	20	9	1/1 | 0	>60s	E S F R
Claude Next.js cleanup	[43]	Deletes project cache	2	0	1/1 | 0	0.11s	E S
Broken file replacement	[8]	Typo truncates file	3	0	1/1 | 0	0.09s	F
File check	[84]	Unset `$bar` used	5	0	1/1 | 0	0.05s	E
Archive extract move	[34]	Moves file to missing dest	11	1	1/1 | 0	0.15s	E S F R
Case-only bulk rename	[73]	No destination `mv`	1	0	1/1 | 0	0.02s	E S F R
Confused mkdir output	[37]	Assumes `mkdir` path output	3	0	2/2 | 0	0.07s	S R
Claude build cleanup	[46]	Failed `cd` to `rm -rf *`	1	2	1/1 | 0	0.07s	E S
ShellCheck author example	[114]	Hidden `rm -rf /`	2	0	3/3 | 0	0.16s	S F R
mkdir output as path	[16]	Captures `mkdir` output	8	0	2/2 | 0	0.21s	S R
Squid init script	[133]	Unknown `rm` target	139	5	4/4 | 0	>60s	S R
Ubuntu backup manager	[20]	Bad `$?` check	766	10	1/1 | 0	>60s	S R
Node.js version manager	[138]	Deletes `/usr/local/*`	247	8	2/2 | 0	>60s	E S F R
iTunes updater	[135]	Deletes drive	4	1	1/1 | 0	0.13s	E S
Steam updater	[134]	Failed `cd` to `rm /*`	450	7	1/1 | 0	>60s	E S F R
NVIDIA driver installer	[131]	Deletes `/usr`	662	13	1/1 | 0	>60s	S F R
ActualBudget updater	[49]	Uses dir after moving it	53	2	1/1 | 0	2.35s	S F R
Repo archiver	[44]	May `cd` into regular file	30	8	1/1 | 0	0.47s	S F R
SLURM cluster overwrite	[128]	Data loss from `xargs mv`	3	0	2/2 | 0	0.28s	F
TV transcode move loop	[124]	Move from deleted dir	10	1	1/1 | 0	0.87s	E S F R
SAS script generator	[93]	Generated file overwrite	7	0	2/2 | 0	0.08s	E F
Hive DROP log monitor	[88]	File overwrite	10	2	1/1 | 0	0.45s	S F
Contest winner mover	[89]	Renames to same file	11	0	2/2 | 0	0.42s	E S F R
Backup rsync script	[87]	Constant `if`	10	0	1/1 | 0	0.12s	E

Continued on next page

Table 5 (continued)

Script name	Source	Bug description	LoC	↓	D/#B | FP	t	$\mathcal{F}$
Two-file cleanup	[95]	Comma not in `IFS`	7	0	1/1 | 0	0.04s	E
Directory chmod loop	[90]	Disabled glob	28	2	2/2 | 0	0.42s	E
MongoDB backup script	[137]	Typo causes DB loss	15	0	18/18 | 0	0.40s	E S R
Zsh installer check	[48]	Use missing shell	90	13	1/1 | 0	>60s	S R
Git version check	[51]	Undefined function	90	0	1/1 | 0	7.56s	S
TIB deploy script	[47]	Copies missing file to itself	22	2	1/1 | 0	0.39s	S R
Git config check	[53]	Unreachable `$?` branch	21	0	1/1 | 0	1.75s	R
Camlp5 install wrapper	[55]	Unset path to `rm -rf /`	205	0	2/2 | 0	8.94s	E S R
Vim config backup script	[50]	Unset var used in test	38	0	1/1 | 0	>60s	E
Debian debootstrap	[9]	Empty `$2` to `cwd` loss	405	31	1/1 | 0	>60s	E S F R
VSCode patch helper	[54]	File check on unset var	12	1	1/1 | 0	0.18s	E
NVM install downloader	[52]	Unset var used for download	137	3	1/1 | 0	13.42s	E
Debian debootstrap 2	[120]	Deletes supplied path	602	45	0/1 | 0	>60s	E S F
OpenStack log collector	[82]	Abort check from `set -u`	36	1	1/1 | 0	>60s	E
Audio watermark build	[125]	Var self-append break	12	0	1/1 | 0	0.15s	E
Total		**Arith. mean (~)**	~71.8	~2.9	115/116 | 0	~14.08s	32E 44S 28F 29R

B Bugs Found in the Wild

Table 6. All bugs SaSh found in the wild two weeks before submission. Each row lists an open-source project in which SaSh identified a bug which we then later manually confirmed and reported to the maintainers, along with the script domain, a high-level description of the bug(s), and the number of bugs found in that project.

Project	Domain	Bug description	#B
AFFiNE [122]	Updater	Dead status handling blocks version updates.	1
Base Node [3]	Setup	Unset-var guards abort node startup setup.	4
BashReduce [31]	CLI	Unquoted input broadens `rm` target.	2
Batocera Linux [4]	Init	Wrong redirection overwrites accumulated logs.	1
CS-Notes [62]	Docs	Undefined variables abort note-build automation.	1
Caker [38]	Build	Misquoted `sudo rm -rf` can remove wrong paths.	1
Cosmos Omnibus [79]	Entrypoint	Unquoted cleanup path broadens `rm` target.	1
Crawl4AI [19]	CI	Status check is dead under `set -e`.	1
Danghuangshang [143]	Daemon	Missing-command assumptions break later setup logic.	1
Dotfiles [41, 80]	Setup	Shell option misuse and inverted tool checks break setup and uninstall flows.	4
Edeliver [29]	Deploy	Broken multi-host validation blocks deployment startup.	1
Embree [106]	Build	Unquoted paths broaden build-script arguments.	1
FaceDetection-DSFD [121]	Dataset	Optional download paths break dataset setup.	1
Ghorg [40]	CI	Dead failure checks hide CI cloning errors.	1
Gloo Gateway [22]	Uninstall	Typo makes uninstaller cleanup dead code.	1
Hasor [14]	Setup	Unquoted directory changes break setup on whitespace paths.	1
IPinfo CLI [63]	CLI	Unquoted self-path and root variables broaden cleanup targets.	2
Kubernetes Test Infra [66]	CI	Wrong array iteration skips parts of the test matrix.	1
La Capitaine Icon Theme [108]	Theme	Mistyped variables break icon-theme maintenance.	1
LibreELEC [70]	Updater	Dead status checks hide driver update failures.	3
Moby [75]	CI	Unset-var flag handling breaks test selection.	2
Multigres [77]	Tooling	Missing arguments break tool-wrapper command construction.	1
Netdata [78]	Monitoring	Broken dry-run error handling hides helper failures.	1
Next.js [141]	Deploy	Unset-var fallback is dead under `set -u`.	1
OpenSC [83]	Build	Missing option values can trap argument parsing in a loop.	1
Openpilot [15]	Setup	Unset-var setup checks abort environment setup.	2
P4 Compiler [7]	Build	Unquoted cleanup paths broaden removal targets.	12
PgBouncer [100]	Test	Edge-case argument handling traps SSL tests in a loop.	1
PlantsVsZombies Fan Game [32]	Build	Mistyped variables break binding generation.	1

Continued on next page

Table 7 (continued)

Project	Domain	Bug description	#B
PyTorch [102]	CI	Unset-var fallback aborts environment setup.	1
RapidPro Docker [119]	Uninstall	Unquoted `pwd` broadens cleanup target.	1
Serverless [111]	Installer	Empty capture breaks installer branch selection.	1
SourcererCC [76]	Tooling	Whitespace in paths breaks shell-script orchestration.	1
SteamTools [5]	Utility	Unquoted paths broaden maintenance-script arguments.	1
Swiftenv [39]	Setup	Unquoted build cleanup broadens removal targets.	1
Tazpkg [118]	Package	Wrong `mktemp` kind makes `cd` fail.	1
Theme Switcher [99]	Configuration	Inverted tool check blocks theme update.	1
ToolSave [27]	Uninstall	Unquoted uninstall path broadens `rm` target.	1
V2M [30]	Uninstall	Whitespace in path broadens `rm` target.	1
Ventoy [140]	Boot	Stale state makes file checks and copy logic misfire.	2
Whishper [101]	Installer	Existing destination directories crash installer setup.	1
ch32-data [12]	Build	Unquoted paths broaden generator-script arguments.	1
vLLM [142]	CI	Indirect status checks make failure handling dead.	4
Total			70

C Symbolic Expansion of All Mechanisms

SaSh expands all seven expansion mechanisms of the shell [58, 60]; this section describes the five omitted in §5.2.

Tilde expansion (~): SaSh expands ~ to the value of `$HOME` in its current environment. `$HOME` is initialized in SaSh's starting state to a symbolic variable (*Cf.* §3.2), with a field count of at least one. SaSh does not currently support the less-commonly-used `~USER`, though it could trivially using the same method with distinct symbolic variables per user mentioned.

Command substitution (`$( )`): SaSh expands command substitution by interpreting the inner commands, and inspects which commands it encountered in doing so to determine the expansion result. If all commands it encountered have known outputs on `stdout` according to their specifications (*Cf.* §4.1), then SaSh expands the substitution to the concatenation of those known outputs. Otherwise, the substitution expands to a fresh, unconstrained symbolic variable.

Arithmetic expansion (`$(( ))`): SaSh does not interpret arithmetic expansion at all; it expands it to a fresh symbolic variable, constrained to a field count of `[1, 1]` (because it must be numeric). In general, arithmetic expansion might mutate any variable, but SaSh does not model these possibilities in line with its optimistic symbolic execution model.

Pathname expansion (globbing): SaSh expands globs into fresh symbolic variables, constrained with a field count of at least one—a glob such as `*` that matches nothing does not expand to nothing but rather expands to itself, so globs always expand to at least one string. While pathname expansion in the shell actually occurs after field splitting, SaSh performs this expansion step before field splitting because it does not query the filesystem.

Quotes and escaping: SaSh expands quoted words according to the shell's semantics, suppressing expansion of other control codes as necessary, and removes outer quotes from the final strings.