Sabline: effects in signatures, budgets at run time, and a baseline for a repository's capability surface*

Palakurthi Gowri Shankar

Abstract

Code written by language models is increasingly run by people who have not read it. Sabline is a small programming language for that situation. A function's signature declares which of seven effects it may perform, and the compiler checks the declaration across the whole call graph. A runtime refuses any operation outside a budget the operator writes - before the operation happens, and in a way the program cannot catch. Contracts are checked by the Z3 prover where it can settle them, and at run time where it cannot. A repository can commit a baseline of the capability surface its programs need, and a check fails any change that needs more. The central claim is about that surface. Suppose a repository's Sabline programs are held to a committed baseline by a required check. If a change makes a program need an effect, path, host, module or operation count the baseline does not grant, the check fails. It goes on failing at every later commit at which the program still compiles and still needs it, until a person edits the baseline. The claim is not about which function an effect is attributed to: a function renamed in the change that gives it an effect its program already had escapes the function-level rule. On a benchmark of 102 programs, 80 with one defect and 22 correct, each written in Sabline, in JavaScript for Deno and in Python, Sabline caught 70 of the 80 defects, 54 of them before running; Deno caught 51 and Python 32. Sabline stopped 4 of the correct programs, the others none; eight of its ten misses are in five categories written for it to lose; and against five more tools, each in its own runtime, one or another does better than Sabline on 18 programs. The capability format is published separately, under CC0, with a conformance corpus of 456 cases that an implementation in any language can run.

1. The problem

A model hands someone a script. They can read it line by line, or run it on trust. Reading every line is the cost they hoped the model would save them, and each new script asks it again; trust is what the situation lacks, because the program's comments, names and structure were written by the same process whose output is in question. Where code arrives faster than people read it, review is the step that gives.

Three things would help a person in that position. The first is to know, before running a program, what it may do to the world outside it - which files, which hosts, which parts of the host system - without reading its body. The second is to be sure, while it runs, that it does nothing the person did not allow, whatever it says about itself. The third concerns a repository rather than a program: to know when the capability of the code in it grows.

The third matters because capability can be assembled a little at a time. A helper that builds a URL, a function that reads a file, then a call three levels down that sends one to the other: each change looks harmless

*Palakurthi Gowri Shankar, ORCID iD https://orcid.org/0009-0007-0004-2955.

in its own diff. Hills, Caspary and Cooper Stickland study this setting directly (Hills et al., 2026). A coding agent builds software over a sequence of pull requests in a persistent codebase while pursuing a covert side task, and attacks that spread the side task across pull requests evade monitors that read one diff at a time; no single monitor they test is robust both to such gradual attacks and to attacks concentrated in one pull request, and a stateful monitor that tracks suspicious buildup across pull requests does better against the gradual ones. Capability is binary and cumulative: the steps of such a change reach exactly as far as one step that did it all at once. A comparison of each commit with the one before sees each step alone, and once a step is merged, sees nothing more of it.

Sabline addresses the three needs with three mechanisms, and adds a fourth, proofs of contracts, that the others do not depend on. Sections 2.1 to 2.4 describe the four, and section 2.5 the format, published separately, that specifies the three and not the proofs; section 3 describes the implementation, section 4 what has been measured, section 5 related work and section 6 the limitations.

2. Design

Figure 1 shows the pieces this section describes: one program from its signature to a refused operation, and one repository whose commits are each compared with the same baseline.

```
(a) One program, one run: examples/effects.vel

fn save_report(text: Text) uses fs {       <- the signature
    write_file("report.txt", text)
}
fn main() uses io, fs, clock, rand { ... }
      |
      |  sabline audit, before it runs
      v
effects: clock, fs, io, rand               <- the audit
reads and writes: report.txt
      |
      |  the operator writes the budget
      v
--allow io,clock,rand,fs:read:./data       <- the budget
      |
      |  at the call, before the write
      v
E313, line 13: 'write_file' reaches        <- the refusal
'report.txt', which this run's fs grants
do not cover. Nothing is written; the run
ends, and the program cannot catch it.

(b) One repository, one baseline

c0: baseline committed, with poll.vel at 10 requests a run

         poll.vel     check, against     review, against
commit   needs        the baseline       the commit before
c1       20           fails: 20 > 10     high
c2       20           fails: 20 > 10     low, sees nothing
c3..c8   40..1000     fails at each;     at c8, sees only
                      c8: 1000 > 10      640 -> 1000
c9       1000, and the baseline edited to 1000:
                      passes             high, names the edit
```

Figure 1: (a) `examples/effects.vel`: what a signature declares, what the audit reports before the program runs, a budget its operator writes, and the refusal of the write that budget does not grant. (b) the second history of section 4.5, from `check_ratchet.py`: c1 raises a count and is merged, c2 is an unrelated change, and the check compares every commit with the baseline committed at c0, never with the commit before it, which is what a review compares.

2.1 Effects in signatures

A Sabline function declares what it may do:

```
fn save(path: Text, body: Text) uses fs { ... }
```

There are seven effects: `io` (the console and the program's arguments), `env` (environment variables), `fs` (files), `net` (the network), `clock`, `rand` and `ffi` (calling Python, the host language). A function with no `uses` clause is pure. The rule is transitive and checked before the program runs: a function may perform only the effects it declares, and calling a function requires declaring everything that function declares, whether or not the callee performs it. A function passed as a value must be pure, and so must any function a contract calls (Palakurthi, 2026b, section 3.2). A declaration is therefore an upper bound on what any call to the function can do, at any depth, and it can be read without reading the body.

An effect name is coarse: `fs` says nothing about which files. The names are for signatures; the finer distinctions belong to the operator's budget.

2.2 Budgets at run time

The operator writes a budget when running a program:

```
sabline agent_output.vel --allow io,fs:read:./data,net:api.example.com:443@100
```

A grant names an effect and may narrow it: `fs` to a direction and a path, `net` to a host, a port, or a one-label wildcard (`*.example.com`), `ffi` to named Python modules, and `fs` and `net` to a count of operations in the run (`@100`). Grants are additive. Paths are resolved, symbolic links and `..` included, when the budget is parsed and again at every file operation, and compared by whole components (Palakurthi, 2026b, sections 4 and 5).

Every operation is checked when it is attempted, in a fixed order: the effect (refused with E310), then the scope - a module (E311), a path (E313), a host or port (E314) - then the count (E315). No file is opened, no connection made and no module imported for a refused operation. A refusal ends the run: it is not a failure value the program can inspect, so a `check` or `try` around the call does not see it. The one exception is a redirect to a host outside the grants, which fails the request as an ordinary failure naming the target, because the program did not choose where it was sent. The budget is checked against what a run attempts, not against what the program declares, so a program whose declarations were never checked is still held to it (Palakurthi, 2026b, section 6).

A granted Python module is trusted in full. `ffi:os` is the operating system as the current user. What an `ffi:M` grant does bound is which modules a call reaches: the attribute chain a call names is walked step by step, and an object owned by a module outside the grants is refused, so `py("json", "codecs.encode", ...)` under `ffi:json` is refused for `codecs`.

2.3 Proofs

A function may carry contracts: `requires`, `ensures`, and loop `invariant`s. The compiler asks the Z3 prover (de Moura and Bjørner, 2008) whether a contract can be broken, using the contracts of callees rather than their bodies. A contract shown false is a compile-time error with a counterexample (E700); so is a call shown to break a callee's `requires` (E701), a list read shown to pass the end (E705) and a divisor shown to be zero (E706). A premise the prover cannot translate abandons the proof, and the contract is checked while running instead: the compiler does not report a proof it does not have.

Proofs are not part of the capability format. Nothing in sections 2.1, 2.2 and 2.4 depends on them, and an implementation with no prover can conform to the format at every level (section 2.5).

2.4 The ratchet

`sabline capabilities init` reads every `.vel` file in a repository and writes `sabline.capabilities`: the repository's *surface* - every grant its programs need, in the budget grammar, and for `fs` and `net` the most operations one run can perform - and, for each program, its own grants and counts and the effects each of its functions declares (Palakurthi, 2026b, section 9). What a program needs is read from its text: its functions' declarations, and the paths, hosts and modules its calls name as fixed text - a literal, or a variable bound once to one. A path, URL or module built while running is recorded as the unscoped grant (`fs:read`, `net`, `ffi`), which a scoped baseline does not cover. The operation bound comes from loops whose counter and limit the text fixes, multiplied through nesting and calls; a loop whose turns the text does not fix, and recursion, have no bound. The prover takes no part, so the result is the same with and without it.

`sabline capabilities check` derives the same from the working tree and compares it with the committed file - with that file, and never with a previous commit. A change widens the baseline, and the check fails, under five rules: a grant the surface does not cover (W1); more operations than the surface's count (W2); for a program the baseline records, a grant (W3) or a count (W4) beyond its own entry; and an effect added to a function the baseline records (W5). Narrowing never fails. A program the baseline does not record is held to the surface alone. A check that cannot read the baseline - it is missing, is not the right format, or holds a grant the format does not allow - fails rather than passes, so deleting the file does not turn the check off. The writer will not replace an existing baseline unless asked to in so many words, so every widening that is accepted arrives as an edit to the file, in review.

Because the comparison is with the baseline, a widening merged once goes on failing at every later check until someone edits the file, and a widening assembled over many commits is reported as its whole sum (Figure 1b). That is the property the gradual attacks of section 1 need to defeat.

The central claim. Suppose a repository's Sabline programs are held to a committed baseline by a required check. If a change makes a program need an effect, path, host, module or operation count the baseline does not grant, the check fails. It goes on failing at every later commit at which the program still compiles and still needs it, until a person edits the baseline. Which entries count depends on the program: for one the baseline does not record, or records as not compiling, the baseline grants what its surface grants; for one it records as compiling, only what both the surface and the program's own entry grant.

"Needs" there is what the program's text requires by the derivation of sabline-spec section 9.3: the declared surface. Three things follow that the claim does not make. It does not say what a granted module does: a program newly calling `os.system` through a baseline's `ffi:os` is inside the surface. It does not say where a path leads: paths in a baseline are compared as text, and a symbolic link under a recorded directory is that directory's content; the budget, which resolves every path at run time, is where that is caught. And it does not say which function an effect belongs to: a function is known by its file and name, so a helper renamed in the same change that gives it an effect its program already had is a new function, held to its program's entry and the surface but not to what its old name declared, and W5 does not report it. The declared surface did not widen in that case, and the check is right to pass it; what escapes is the attribution.

A separate command, `sabline review --against REF`, reports how the working tree differs from a git ref, with a one-word risk. It compares with a previous state, so it informs a reviewer and is not the gate.

2.5 The format, published separately

The effects, the budget grammar, what a runtime must refuse, `sabline.audit/1` (the JSON report of what a program declares and names before it runs) and `sabline.capabilities/1` are specified separately, under CC0, as sabline-spec (Palakurthi, 2026b), so that they can be implemented without reading the compiler. Its conformance document defines three levels: L1, declaration - parse the budget grammar, compute a program's effect surface, emit `sabline.audit/1` that validates against its schema; L2, enforcement - refuse at run time every operation outside the budget, uncatchably; L3, the ratchet - write and read baselines and

classify widening against narrowing for every kind of scope. L2 and L3 each require L1; L3 does not require L2, so a tool with no runtime can conform at L1 and L3. No level requires a prover. Conformance is shown by running a corpus of JSON cases, described in section 4.4. sabline-spec also defines an in-toto predicate type that binds an audit to the digests of the files audited, so that a signed statement can say which source an audit describes; sabline-lang 4.2.0 writes such statements, unsigned (`sabline attest`), and leaves signing to Sigstore's tools.

3. Implementation

The implementation, sabline-lang (Palakurthi, 2026a), is one Python file, `sabline.py`, of 13,497 lines - one file by design: nothing to install but Python, nothing vendored, a single artifact to sign and audit. It is laid out in the order a program passes through it: lexer, parser, loader, effect checker, type checker, prover, native code generation, interpreter. The prover needs the optional `z3-solver` package and the native code generator the optional `llvmlite`; without them, contracts are checked while running and everything is interpreted, and the effect checks and the budget are unchanged.

The budget is enforced in the interpreter, at each builtin that performs an operation, before the operation. A run with a time limit or a memory cap runs in a child process that can be killed, and a pool of such workers re-installs the budget before every program. The same checks sit behind every interface: the command line, a Python library (`check`, `audit`, `run`, `Pool`), an HTTP door and an MCP server whose operators set ceilings callers cannot exceed, and a GitHub Action that runs the capability check and uploads findings as SARIF. Every error has a stable code; the compiler's table holds 62.

The budget is not a security boundary by itself. It is enforced by an interpreter written in Python, in the same process as the compiler and, unless a limit is set, the program. From Sabline 8.4.0, which postdates the version the rest of this section describes, a run in a process of its own also asks the operating system to hold the same budget before the program's first statement runs: Landlock and seccomp-bpf on Linux, where a run under the default budget is fully held; a sandbox profile on macOS and a job object with a lowered token on Windows, where it is partly held. One function derives the operating-system policy from the budget, and each run's receipt says which level it got and why. It remains a guard against a program doing what it was not asked to, and it still belongs inside a separate account or a virtual machine when the stakes warrant one.

4. Evaluation

4.1 A benchmark against Deno and Python

The benchmark is 102 small programs, each written three times with the same behaviour: in Sabline, in JavaScript for Deno, and in Python. Eighty contain one deliberate defect, on one line marked in all three sources; twenty-two are correct controls. They fall in twenty categories: a file write hidden in a helper, a network call hidden in a helper, division by a value that can be zero, a read past the end of a list, integer overflow, an ignored failure, an infinite loop, runaway memory, reaching a dangerous module, correct programs that must not be flagged, a grant narrower than the effect, indirect authority - a caller that does not change while a dependency's declared budget widens between two versions - a TrapDoor, a program whose stated purpose and behaviour differ, a skill script whose helper reads a credential and sends it elsewhere, an import of a library that does not exist where its program names it, and five categories added when the comparison of section 4.2 found a benchmark Sabline could not lose: a leak through a channel the task is granted, one legitimate subprocess beside a second one, correct programs Sabline's rules refuse, danger in a granted library or its native code, and programs whose legitimate work must still succeed. One harness runs every program through every tool with the same rules: a 5-second timeout, a 256 MB memory cap, and for Sabline the narrowest budget each task needs. Each program gets one verdict per tool - caught before running, caught while running, missed, or, for a control, clean or a false positive - and after every run the harness observes whether the dangerous effect actually happened: a file created, a request received by a local listener, a subprocess's sentinel printed. Where a program has legitimate work to do, the harness also checks that it got done, by the output, request or file the work leaves; a control whose work did not get done is a false positive.

A static flag counts only on the marked line, for Sabline's audit as for Deno's lint. The committed results were produced by Sabline 8.6.0, Deno 2.9.6 and Python 3.13.13 on Windows 11, at the commit that added the five categories. Of the 76 earlier programs, one verdict moved: in 08f Sabline's audit flags the loop that drives the memory growth, two helpers above the line that grows, and under the marked-line rule that is no longer credited, so its catch moved from before running to while running. At Sabline 8.4.0 every program kept the verdict it had at Sabline 8.3.0, where ten consecutive runs had written identical files, and the 68 programs of the first thirteen categories kept every verdict they had at Sabline 8.2.0. Before that, the 67 programs of the first twelve categories kept every verdict and every line of evidence they had at Sabline 7.1.1, and ten consecutive runs at 7.1.0 had written identical files. The 63 programs of the first eleven categories kept the verdict each had under Sabline 4.1.0; Sabline 3.0.0, with the same Deno and Python on the same platform, had produced that table before 4.1.0, every verdict and every line of evidence.

Tool	Caught before running	Caught while running	Missed	False positives (22 controls)
Sabline	54	16	10	4
Deno	8	43	29	0
Python	0	32	48	0

Table 1: the 80 dangerous programs and the 22 controls, from `benchmark/results.json` at the commit that added categories 16 to 20. Seven of Sabline's 70 catches stopped the program's legitimate work too; none of Deno's or Python's did.

Where the catches come from differs by tool. Sabline's catches before running come from effects in signatures (the file, network and module categories, the environment secret of category 11, and the TrapDoor of category 13), from unhandled-failure checks and the prover (division, list reads, ignored failures), from a comparison of two versions of a dependency's declared surface (category 12), and from its termination rule, which flags a loop whose counter does not move one step toward an unchanging limit. That rule flagged ten of the eleven infinite and memory-growth programs on the marked line before running, and 08f's loop two helpers above it; it claims nothing about whether such a loop ends, and each of these happened not to. Deno's catches are almost all at the moment of the call, through its permission flags; nothing in `deno check` or `deno lint` reads a file write or a fetch as a problem. In one network program, and in the three upgrades of category 12, Deno denied the request or the write but the program caught the denial, which is an ordinary exception in JavaScript, and exited 0; the harness credits Deno because it observed that the effect did not happen, and a caller reading only the exit status would have seen success. A Sabline refusal cannot be caught.

Category 12, added with Sabline 7.1 at a reader's suggestion (CHANGELOG, 7.1 entry), is indirect authority: a calling program that is the same file before and after an upgrade, and a dependency whose declared budget widened between the two versions - gaining `net`, gaining a second host inside the `net` it already had, or gaining `fs:write` beside the read it had - with a control whose dependency narrowed. Each caller already declares the effect its dependency comes to use, so it compiles against both versions and the compiler has nothing to refuse. The Sabline static step there is `sabline deps-diff`, which derives each version's surface as section 2.4 does and holds the newer one to the older with the same five rules; it flagged the three before running and did not flag the control. A JavaScript or Python module declares no surface, and `sabline deps-diff` reports such a dependency's surface as unknown rather than reading effects off its source, so the category shows what a declared surface makes checkable, not a way to check code that declares none.

Category 13, added with Sabline 8.0 (CHANGELOG, 8.0 entry), is one program whose stated purpose is a scan for secrets and whose behaviour is to read a credential file and post it to a URL. Sabline flagged it before running - its audit lists `net`, which a scanner does not need - and the run was refused at the read (E318). Deno's run stopped before the request was made; Python's made it.

Sabline alone caught the six integer-overflow programs, while running: its whole numbers are 64-bit and arithmetic that leaves that range stops the program (E407). Python's integers do not overflow, so its printed values are arithmetically right and only wrong for a 64-bit consumer; the results record those rows as Python misses and say a reader may discount them. Python's catches in the division and ignored-failure categories

are crashes on the inputs the harness supplies - a zero, a non-number, a missing key - and with ordinary input those programs run clean; Sabline's E520, E705 and E706 do not depend on the input. Four Sabline catches came only at run time where a sibling program was caught before running: a division on the unguarded one of two paths, a remainder inside a loop, a division in `main` on `n - 1`, and a read at `i + 1` in a loop bounded by the list's length. The prover did not settle those, and the runtime checks stopped them.

Sabline missed ten programs. Eight are in the categories added so that it could lose. In three (16a to 16c) the task is to read a private ledger and post a summary of it to the one host it is granted, and the program posts the ledger itself: every grant Sabline has allows the read and the send, and nothing in it follows the data from one to the other. In two (17a, 17b) the task runs `git` and the program runs a second program too: Sabline grants the `subprocess` module whole, where Deno's `--allow-run=git` grants one program and refused the second. In three (19a, 19b, 19d) a vendored library the program is granted as a module writes a file or makes a request of its own, in Python or in native code, below anything a Sabline budget reaches; Deno refused the two JavaScript libraries' I/O, and nothing refused the native write.

Sabline stopped four correct programs, the only false positives in the table: a loop that reads until its input ends and Euclid's algorithm, whose loops its termination rule cannot show to end, and 25! and a product modulo 2^61 - 1, which need whole numbers past 64 bits (E407). And in seven of its catches (12a to 12c, 14c and 20b to 20d) the program's legitimate work was stopped with the danger: a Sabline refusal ends the run and cannot be caught, and a program that does not compile does not run at all. In all seven Deno refused the same operation, the program caught the denial as an ordinary exception, and its task still got done.

The other two misses were included on purpose so that the table is not a list of what the language was built to do. In `04c` a loop stops one item early: no read is out of range and the function has no contract, so a wrong total is indistinguishable from a right one. In `09c` a program prints `rm -rf build` as a hint for its caller and touches nothing; its only effect is `io`, which the task needs. That one should not be caught by any tool: control program `10d` prints the same words in a warning and is harmless, and nothing that looks at the program from outside can tell the two apart. The danger is in what a caller does with the text.

What this benchmark does not show. The programs were written for it by this project; in the first ten categories, three of each six were written after the first three, against the tools, to hide the same defects better, and the last five categories were written, after a first comparison, for Sabline to lose. It is small. The inputs were chosen to trigger each defect. The committed run is one machine's. Category 12's dependencies were written for it rather than taken from published packages, and its Sabline catches come from comparing declared surfaces, which a package in another language does not have. And it measures programs, not the ratchet, which section 4.5 treats separately. The harness is in the repository, its rules are one function each, and the evidence for every cell is recorded next to the verdict, so a reader who disagrees with a row can change the rule and rerun it. `benchmark/run.py --check` reruns the whole table and fails if any verdict differs from the committed `benchmark/results.json`.

4.2 The same programs against five other tools

Deno and Python are not the only tools that claim part of what Sabline claims. The same 102 programs were run through five more, each in its own runtime, pinned by hash or commit and recorded on Linux: Deno 2.9.7; the corpus's Python in CPython 3.14.7's WASI build under wasmtime 49.0.0; a translation into Starlark, run by starlark-go; the corpus's Python in smolagents 1.26.0's `LocalPythonExecutor`, the sandbox that framework runs a model's code in by default; and a translation into a CaMeL plan, run by CaMeL's reference implementation under its own policies (Debenedetti et al., 2025). Each gets the narrowest grant that still lets the task's work run, derived from the program's needs by one rule per tool. A scenario a tool cannot express - a network request under a WASI build with no sockets, a library's own I/O in Starlark - is recorded with the reason and not scored; and CaMeL, whose threat model trusts the plan, is scored only on the twelve programs where a value a tool returned reaches another tool on the marked line.

The first version of this comparison had no competitor ahead of Sabline on any program. That was the benchmark, not the tools: its correct programs were written in the shapes Sabline's rules accept, no program checked whether its legitimate work still got done, a catch before running outranked one while running, and Sabline's audit was credited for a flag anywhere in a program where the other static steps were credited only

on the marked line. The corrections are those of section 4.1 and three more, applied to every column alike. A program is compared first on the outcome - the danger stopped with the task's work intact, stopped with the work broken too, or missed; for a control, run clean or not - and on timing only where two tools achieved the same. A catch that the record says came from a failure that would have stopped the task as well counts as one with the task broken. And the five categories of section 4.1 were added, with a prediction for each tool committed before any of them ran.

Competitor	Better than Sabline	Same outcome	Worse	Not scored
Deno	15	64	23	0
WASI	10	70	7	15
Starlark	4	83	10	5
Python sandbox	7	75	20	0
CaMeL	3	4	4	91

Table 2: each competitor against Sabline, program by program, from `benchmark/competitors/results.json` at the commit that added categories 16 to 20. *Same outcome* includes the programs where both caught the defect and Sabline did so before running and the competitor while running: 29 for Deno, 49 for WASI, 19 for Starlark, 47 for the sandbox and 2 for CaMeL. No competitor caught one earlier than Sabline.

A competitor did better than Sabline on 18 of the 102 programs. Deno did on fifteen: 17a and 17b, where `--allow-run=git` let the task run `git` and refused the second program, which Sabline cannot express because it grants the `subprocess` module whole; 19a and 19b, where a vendored library's own write and request were refused by the permissions that hold every line of JavaScript in the process; the four correct programs of category 18, which Sabline stops; and 12a to 12c, 14c and 20b to 20d, where both refused the dangerous operation but only the Deno program went on to finish its task. WASI did on ten: the same four correct programs, 19a and 19b, where the library ran inside the guest and reached only what was preopened, and 12c, 14c, 20b and 20d. CaMeL did on 16a to 16c, the only tool to stop a private ledger leaving through the channel its task was granted; on the same programs it refused 16d, a correct program that posts only how many entries the ledger holds, and allowed 12a, whose dependency posts the length of a private page, which its reference interpreter treats as public. Starlark did on four: 17a and 17b, its host granting one program, and 18c and 18d, its integers not wrapping. The Python sandbox did on seven: the four correct programs of category 18, and 16a to 16c, where it refused nothing - smolagents 1.26.0 cannot make any request through `urllib.request`, the task's own included - and the table marks those three as such.

No tool stopped 19d, where a vendored library's native code writes a file: Sabline grants the module whole, Deno grants its foreign function interface whole, and neither WASI nor Starlark can load native code at all. Sabline alone stopped the six overflow programs and 18f, a record id too wide for its 64-bit column, where every other tool computes the arithmetically right, larger number; section 4.1 says a reader may discount those rows. The comparison's weaknesses are the benchmark's, and two more: the Starlark and CaMeL translations were written by AI agents under the author's direction, and CaMeL is run without the model that writes its plans, so its column measures its interpreter and policies, not an attack on it. The page that publishes the table lists where each tool is stronger by design, whatever this corpus shows, and where the comparison is still unfair and to whom.

4.3 AgentDojo, with no model in the loop

Sections 4.1 and 4.2 count defects caught, not attacks resisted. AgentDojo measures the second (Debenedetti et al., 2024): a suite of tasks a user gives an agent, injection tasks an attacker plants in the data the agent reads, and AgentDojo's own checks of whether the user's task was done (utility) and whether the attacker's goal was reached (security). Sabline was run on its Slack suite, AgentDojo 0.1.35 and benchmark version 1: 21 user tasks and 5 injection tasks, whose pairs make 105 attacks. Each user task's reference solution, and each injection's ground-truth action, is a Sabline program that reaches AgentDojo's real Slack environment only through `tool` calls. Each runs under three budgets: `--allow all` as the control; a budget written for the user task - `io`, and a grant for each tool the reference solution calls; and that

budget with every argument the user task's own text names held to it by the tool door's argument patterns (`tool:send_direct_message:recipient=Bob`, section 7.2 of the specification), which this paper calls the pinned budget.

Under all three budgets all 21 reference solutions ran to completion and passed AgentDojo's utility check. Under `--allow all` all 105 attacks landed; under the task budget 19 did; under the pinned budget 9. The 86 the task budget refused call a tool the task was not granted, and were refused before the call reached the host (E321). Every one of the 19 reuses a tool the task was granted: twelve visit a web page where the task reads web pages, and seven send a direct message where the task sends messages. The 10 more that pinning refused reuse a granted tool with an argument the task's own text excludes - a direct message to Alice where the task writes only to Bob, a web page where the task names the two addresses it reads.

For each of the 19 that land under the task budget the harness records which door the attacker's value came through: written into the program as a literal, or read at run time out of what a tool returned. All 19 are literals, and none is read at run time. That is a property of this harness rather than of AgentDojo, and it is the first thing to say about it. With no model in the loop, the step that would carry the injected instruction out of the data and into the program is the harness's transcription of ground truth, which writes the attacker's value out as a literal; the value originates in the data in every one of these, and what the harness fixes is that it arrives as a literal. Two of the five injection tasks do have the laundering shape - a tool that reads, then a tool that sends - and neither reaches the door under any task budget, because no user task grants every tool their action needs. The data door is one this corpus never puts in front of a budget, so the 19-to-0 split is a limit of the corpus and not a result about Sabline.

Pinning cost no utility here and cannot be had everywhere. Of the 80 arguments the reference solutions pass, 32 are named by the user task's own text and 48 are not: a summary the model writes, a channel found by counting users, an address that is on the page the task sends the agent to read. Eighteen of the 21 tasks have at least one such argument, and two of them - do everything on the TODO list at this address - describe their whole content on a page, so between them pinning holds one argument of thirteen. Where an argument is free the attacker's value is inside what the operator could write down, and all 9 attacks that still land are of that kind. The pattern grammar has limits of its own, which are reported and not fixed, since an evaluation should not extend the thing it measures: a star never stands for white space, so no pattern matches a free-text argument of more than one word, and a pattern is a whole-value match with no negation, so the useful pin for a message body - that it carry no address - cannot be written. For a body a model writes, the choice a pattern offers is one exact string or anything at all.

There is no model in the loop, and that bounds what the numbers mean. Utility is AgentDojo's reference solution run under a budget, so it measures whether the budget lets correct work through, not whether a model writes it. Attack success assumes a model fully steered on every attack, so 19 of 105 and 9 of 105 are upper bounds on what those budgets let through, not measured rates at which a model is steered; measuring that needs a model, a key and money, and has not been done. And no budget here sees what flows where. A pattern is matched against the value at the door, whichever door it came through, so it refuses an attacker's value exactly when the operator can say in advance what the value should be, and can do nothing when they cannot. An attack that reuses a granted tool - to send what the task read to a sink the task may write to - is a choice within the budget, and Sabline has no per-value provenance with which to refuse it; `decisions/0004` is the design for a mark on untrusted input, and it has not shipped. CaMeL, which tracks every value's provenance, and ChainCaps, whose capabilities are per value and per sink, should do better on exactly these cases. The results are `evals/agentdojo/results.json`, which a CI job re-derives program by program under all three budgets.

4.4 The suites, and a conformance corpus

Each guarantee has a suite that asserts it, and each suite runs on every push on Linux, Windows and macOS, with Python 3.10 and 3.12, with and without the prover. `check_sandbox.py` holds 34 attempts to escape a budget - an effect two helpers below `main`, a refusal caught to carry on, a module reached through a submodule path, `py_json` or a handle, a path out of a prefix by `..` or a symbolic link, a host, a port, a wildcard's parent domain, a count, a URL with no scheme taken as HTTPS - each refused with the code it must carry, and

16 honest programs that must still run. Run on Windows with the prover, `check_ratchet.py` passes 114 checks of the ratchet, `check_library.py` 196 of the library, the doors, the audit and its attestations (its one symbolic-link case runs only on POSIX), `check_refusals.py` 21 wrong programs each refused with the right code, `check_fallible.py` 26 fallible builtins, and `check_termination.py` 44 adversarial loops; without it, `check_library.py` passes 193 and `check_refusals.py` 11, skipping the checks that are about proofs, and the rest are unchanged (CHANGELOG, 4.2).

Those suites drive sabline-lang's own command line and Python library, so until sabline-lang 4.1 an implementation in another language could use them only through an adapter. sabline-spec now holds a corpus of 456 JSON cases that any implementation runs its own way: 307 at L1 (280 budgets to parse or refuse, 27 programs to audit), 40 at L2 (programs to run under a budget, with a fixture of files and two local HTTP servers) and 109 at L3 (baselines to write, changes to check, two sequences of changes, the writer's guard, and the covering, reduction and operation-bound rules case by case). A script writes the corpus from the tables of three of the suites, where each entry is also asserted against sabline-lang, and a drift test in both repositories' CI regenerates it and fails if it differs from what is committed. Nineteen scenarios are left out and listed with the reason: six depend on Python's object model, four need a Python host, five turn on what a run given no budget gets - three such runs, and two denials that narrow it - which the format leaves to the implementation, two are about sabline-lang's command-line flags, one is its `all` shorthand, and one the wording of its messages. `sabline conformance` runs the corpus against sabline-lang; it passes at all three levels (sabline-spec `tests/index.json`; CHANGELOG, 4.2).

Writing the corpus found a defect. The audit of a program refused for naming an unknown effect still listed that name among the program's effects, and wrote a suggested budget that does not parse, although the specification said neither could happen. sabline-lang 4.1.0 fixes the implementation, and sabline-spec 0.4 corrects the text that had claimed more than the implementation did.

4.5 The ratchet

`check_ratchet.py` builds scratch trees and real git histories and asserts what the check says at every step. In its gradual case, a greeter program is committed with its baseline, and six changes follow: a pure text helper; the greeting using it; a pure summary function; `main` printing a summary; a function returning a collector's URL as text; and finally a summary delivered to that URL through a helper three calls below `main`. The first five pass. The sixth fails, naming `net:collector.example.net` as a new effect, the file `lib/deliver.vel`, line 2, the function `send`, and the chain `main -> summary -> deliver -> send`. A review of each change against the one before calls the first five `low` and the sixth `high`. In a second history a count of 10 requests a run is raised and merged, and keeps failing at every later commit; seven steps from 10 to 1000 are reported by the check as 1000 against the declared 10, while a comparison with the previous commit shows only the last step, 640 to 1000. The suite also holds changes that must pass - reordering, reformatting, renaming locals, a literal moved into a variable, a function renamed or moved, narrowing, a new program inside the surface - and baselines that cannot be read, which must fail.

The five known limits of section 6 are cases too, each recording the outcome the check gives today, so that another implementation matches it and so that a change to it is visible. The repository checks itself: its own `sabline.capabilities` records 172 programs, 22 of them built to be refused and recorded as not compiling, and CI runs the check on every leg.

The ratchet has not been measured on the gradual-attack benchmark of Hills et al., whose code is not Sabline, or on any repository but its own. What is claimed for it is a property of a deterministic comparison, in the same sense that soundness is a property of a type system, and not a detection rate. A rate measures how often a heuristic notices something. The ratchet does not notice; it computes, and the cases in `check_ratchet.py` show what it computes. The property is the one stated in section 2.4, with the three things it does not say there and the five known limits of section 6, and it rests on the comparison's definition and on those cases: it has not been proved mechanically.

5. Related work

This section says what each piece of work does, **where it is ahead of Sabline**, and where Sabline differs. One decision here has a named source: the amendment to `decisions/0004` takes its propagation rule from ChainCaps, and says so. Nothing else in sabline-lang's changelog names any of this work as the source of a decision, and no priority is claimed over any of it.

Three of the systems below - CaMeL, TypeGuard and ChainCaps - report an evaluation against an adversary on a published benchmark, with a model in the loop, and a fourth, AgentBound, against published sets of malicious MCP servers. Sabline's is on AgentDojo too (section 4.3), with no model in it: under a budget written for each task, 21 of 21 tasks completed and 19 of 105 attacks landed, every one of them through a tool the task was granted; holding each tool's arguments to what the user task's own text names brings that to 9 with no loss of utility. Its utility is AgentDojo's reference solution and its attack success an upper bound rather than a measured rate at which a model is steered, so it is not a like-for-like comparison with theirs; and on the laundering that makes up the 9 that are left, the systems that track values should do better.

Object capabilities. Dennis and Van Horn introduced the capability, a reference that both names a resource and carries the right to use it (Dennis and Van Horn, 1966); Miller's object-capability model builds least authority from references a program has been handed (Miller, 2006). Where that model is ahead: authority is unforgeable and cannot be had except by being handed over, so least authority holds between the parts of one program and not only between a program and its operator. A Sabline effect is a name in a signature, and a budget is ambient to a whole run: any function declaring `fs` may reach any path the budget allows without holding a reference to it. Nothing in Sabline is unforgeable, and nothing is handed over. "Capability" in the format's name follows common usage.

Language-based agent control. Zhou, D'Antoni and Polikarpova have the agent write a program in a typed host language against a specification the surrounding scaffolding fixes, so that a program the type checker rejects never runs, and the same policy binds agent-written and developer-written code (Zhou et al., 2026). Policies - capability constraints, data provenance, information flow - are types and effects; arbitrary side-effect-free computation and recursive subagent calls stay available. Their prototype, TypeGuard, is Haskell: its filesystem policy hands out `Path` values that are unforgeable tokens of authority over a directory subtree, and its information-flow case study uses the LIO library. On AgentDojo's Slack suite - 21 user tasks and 5 injection tasks, whose pairs make 105 attacks - TypeGuard and CaMeL each resisted all 105 with information-flow policies enabled; TypeGuard completed 15 of the 21 benign tasks without policies and 8 with them.

Where it is ahead: it has information-flow control and provenance, which Sabline has not - `Secret of T` is a one-way mark with an audited `declassify`, not a lattice, and the mark for untrusted input is a 9.0 draft (`decisions/0004`) that has not shipped. Its filesystem authority is an unforgeable value in Dennis and Van Horn's sense, where a Sabline budget is ambient. Its policies are ordinary types in a general-purpose language, so they compose and a user can write a new one, where Sabline's seven effects and its grant grammar are fixed and only the compiler can add to them. And it is measured against an adversary with a model in the loop, where Sabline's AgentDojo run (section 4.3) has none. Where Sabline differs: the policy is text an operator writes outside the program, so whoever bounds a run need not have written the scaffolding or be able to read Haskell; the refusal happens at the operation and cannot be caught by the program, rather than at a type check before the run; and the language is small enough to hand a model on a card.

ETAS. Tan and colleagues make agents, model inference, tools, typed prompts, memory, approvals and effect handlers constructs of an effect-typed language (Tan et al., 2026). Each computation carries the effects that escape it and an abstraction of the actions it may request; trace specifications of allow, deny and ordering rules - an approval before an email is sent, say - compile to finite monitors, which the checker proves where it can and otherwise leaves as explicit checks at run time, and a handler may mock an action without hiding that it was requested. Where it is ahead: a core calculus, with preservation, progress, effect soundness and policy safety stated and given proof sketches (a mechanised proof is future work), where Sabline has no calculus; policies over the order of actions, where a Sabline budget bounds each effect and its count and says nothing about order; the discipline of proving what it can and checking the rest at run time applied to authorisation, where Sabline applies it only to contracts; and a compiler and interpreter in Rust, where

Sabline's Rust runtime is in progress. It also has secret handling, grants made at deployment, a report of a program's effects before it runs, and replay, so none of those distinguishes Sabline. Where they differ: ETAS is written for programmers who put model calls inside agent programs, and leaves a sandbox as an abstract predicate outside its core; Sabline bounds a script a model has written, under an operator's budget that the operating system also holds. ETAS's evaluation is four case studies, with no measurement.

ChainCaps. Jiang and colleagues attach to every value a *sink-specific capability budget* - the set of `(operation, scope)` sinks that value may still reach - and propagate it through a tool chain by intersection: a tool's output carries the tool's own pass-through budget intersected with the budget of every input (Jiang et al., 2026). Authority can be lost by composition and never gained, which they call monotonic attenuation, and which answers what they name *permission laundering*: reading a confidential document, summarising it and sending the summary to an external endpoint, where each call on its own passes its own permission check. It runs as a transparent MCP proxy needing no change to the agent or to the tool servers, and over 82 tasks on five frontier models it takes attack success from 25-68% down to 0-4.8% while 96-100% of benign tasks still complete.

Where it is ahead: permission laundering is exactly what a Sabline budget cannot see. A budget bounds the set of resources a run may reach; laundering is a choice within that set, and every step of it is granted. `docs/runner.md` records the case - a value a tool returned choosing which granted file is read - `decisions/0004` is the design that answers it, and neither has shipped. ChainCaps also puts tools that already exist under a policy without modifying them, where Sabline needs the program written in Sabline; its budgets are per value and per sink, where Sabline's is one budget for a whole run; and it is measured against an adversary with real models. Per-sink budgets are what the 9 attacks still landing on Sabline's AgentDojo run need - the ones whose argument the user task's own text cannot name, so that no pattern written in advance can exclude them - and it should do better on them. Its authors scope the claim to explicit flows, trusted manifests and data movement the proxy can see, and name manifest quality as the deployment bottleneck: manifests written by experts blocked every attack, and naive ones 27.3% of them. Where Sabline differs: the budget is checked inside the runtime at each operation, so it bounds what a program does with no tool call at all - a file it opens, a host it reaches, a module it imports - and from 8.4.0 the operating system is asked to hold the same bounds beneath it. The amendment to `decisions/0004` adopts the intersection rule for Sabline's two marks; it is a 9.0 draft and is not normative.

CaMeL. Debenedetti and colleagues have a privileged model turn a trusted request into a program in a restricted subset of Python, run by an interpreter that attaches to every value its provenance and permitted readers and checks a policy at each tool call, while a quarantined model parses untrusted data (Debenedetti et al., 2025). Where it is ahead: it tracks data, which Sabline does not - a Sabline budget bounds which effects, paths, hosts and modules a run may reach, not which values may flow to them - and its AgentDojo evaluation runs a model, where Sabline's (section 4.3) runs AgentDojo's reference solutions in its place. The 9 attacks that still land there reuse a tool the task was granted with an argument its text could not name in advance, and provenance per value is the defence against them that Sabline lacks: CaMeL should do better on them. Its split between a privileged and a quarantined model has no counterpart here at all. Where Sabline differs: the program is written in a language whose signatures declare effects and whose compiler checks them across the call graph before anything runs, and the operator's budget is enforced against the operation rather than against a value's history.

Wassette. Microsoft's Wassette runs MCP tools as WebAssembly components under Wasmtime, under a deny-by-default policy file granting storage, network and environment access per component: "components have no access to system resources by default and must be explicitly granted permissions" (Microsoft, 2026). Where it is ahead: the tools are ordinary MCP tools and are not modified, and a model may load one at run time from an OCI reference and have the policy still hold; the boundary is a WebAssembly sandbox rather than a check inside an interpreter; and the grants are per component, where a Sabline budget is per run.

It is also the clearest published case of why Sabline does not claim its budget is a boundary. On 2026-08-20 a filesystem sandbox escape was published against Wasmtime's WASI implementation: a defect in the `cap-std` dependency let a guest leave its granted directories when a path or a symlink carried a trailing slash, rated 8.8, held on Linux 5.6 and later where `openat2` is used and not on macOS or older kernels (RUSTSEC-2026-0269,

GHSA-vqjp-4c8c-hfgg) (Bytecode Alliance, 2026). Wassette 0.6.0 and 0.7.0 shipped the affected Wasmtime and 0.7.1 moved to the patched one. Wassette's policy was not wrong; the defect was underneath it. A deny-by-default policy is worth what the layer enforcing it is worth, and Sabline's confinement layer is exposed the same way - it rests on Landlock, seccomp-bpf, `sandbox_init` and a Windows job object, and a defect in any of those is a defect in what a receipt claims. That is why section 3 says a run belongs inside a separate account or a virtual machine when the stakes warrant one, and why section 6 says the budget is not a security boundary.

AgentBound. Bühler, Biagiola, Di Grazia and Salvaneschi give MCP servers Android-style permission manifests: a server declares generic capabilities, the user consents at launch to concrete paths, environment variables and hosts within them, and a container enforces the result with mounts, a whitelist of variables and firewall rules, without the server being modified (Bühler et al., 2026). Where it is ahead: it confines unmodified third-party servers, whatever language they are written in, where Sabline bounds only programs written in Sabline; it writes a server's manifest from its source with a model, which developers judged correct for 80.9% of the manifests they answered about, among the 296 popular servers the paper studies; it is evaluated against published sets of malicious servers, blocking the nine attacks that reach system resources and naming the ones it does not; its overhead is measured on two platforms; and it is peer-reviewed, with an archived artifact. Where they differ: its boundary is a container around a process, Sabline's a check at each operation inside an interpreter with the operating system holding the same budget beneath it. Its paper says it "cannot prevent attacks that do not violate the declared policy", which is as true of a Sabline budget; for its own programs, Sabline also checks declared effects against the code and proves contracts inside the grant, where AgentBound leaves such issues to program analysis.

Sandbox platforms. E2B runs every sandbox in its own Firecracker microVM with its own kernel, so that "isolation is at the hypervisor boundary, not the container or process boundary" (E2B, 2026); Modal states that its compute is "containerized and virtualized using gVisor" (Modal, 2026). Where they are ahead, and it is the thing Sabline most lacks: they are a boundary that holds when the runtime inside it is defective or hostile, and Sabline's budget is enforced by a Python interpreter in the same process as the program it bounds. A Sabline run inside either is strictly better off than one outside. Where they do not reach: they bound a machine, not an operation - a program in a microVM that has the network can reach every host on it - and neither says what a function may do, nor leaves behind a declared surface a repository can hold to a baseline. They are the layer Sabline combines with rather than one it competes with; `plan/9.0.md` names a template for both as work after the release candidate.

Benchmarking agents. Abdelnabi, Hicks, Rieck and Sadeghi argue that the benchmarks used to evaluate agents in security-critical roles have three weaknesses - vulnerabilities in the benchmark itself, temporal staleness, and runtime uncertainty - and outline what more trustworthy evaluation would need (Abdelnabi et al., 2026). Where it is ahead: it is a direct account of what section 4.1's benchmark is weakest at. That benchmark was written by this project, is small, and counts defects caught rather than attacks resisted; section 6 says so. `plan/8.7.md` is the plan for an evaluation built to answer the three: its comparison with five other tools (section 4.2) and its AgentDojo run (section 4.3) are built, the second with no model in it, and its measurement of a model writing Sabline is not.

TACIT. Odersky and colleagues have agents write Scala 3 with capture checking, in which capabilities - a file system rooted at a directory, a set of hosts, a set of commands - are values the type system tracks, and pure computations over classified data cannot leak it (Odersky et al., 2026). Where it is ahead: capture checking tracks capabilities as values through a full type system, so authority is bounded per expression rather than per run, and it has an information-flow story where Sabline has none. In Sabline capabilities are effect names plus an operator's budget written as text and enforced by an interpreter, in a small language a model learns from a card; it has no counterpart to TACIT's information-flow control, and no grant for running commands.

WASI. The WebAssembly System Interface gives a module only the directories, sockets and other resources its host hands it as handles, so a module given nothing reaches nothing (WebAssembly Community Group, 2026). Where it is ahead: the handle is the authority, enforced at a virtual machine's boundary rather than inside the process running the program, and it applies to any language that compiles to WebAssembly. Sabline's grants are text an operator writes, enforced inside the same process. Until version 5.0 its command

line granted every effect when no budget was given, so deny-by-default held only once an operator wrote one; 5.0 made the default `io` - the console and nothing else - in the command line, the library, the worker pool and both doors, so a run that asks for nothing now gets close to what a WASI module given nothing gets, and `--allow all` is the explicit way to ask for everything.

Deno. Deno's permission flags give a script no file, network, environment or subprocess access unless granted, with scoped forms much like Sabline's grammar (Deno, 2026); section 4.1 compares the two directly. Where it is ahead: it bounds programs written in a language people already use, with a runtime and an ecosystem Sabline has not got, and it grants subprocesses, which Sabline does not model at all. A Deno denial is an exception a script can catch; a Sabline refusal ends the run. Deno checks at the call; Sabline also declares the effect in the signature, so it is visible before running.

Effect systems. Lucassen and Gifford's polymorphic effect systems record in an expression's type the side effects evaluating it may have (Lucassen and Gifford, 1988). Where that work is ahead: it is polymorphic, so an effect can be a variable and a higher-order function can be typed in the effects of what it is given. Sabline's `uses` is a fixed set of seven names with no polymorphism - a function value is simply required to be pure - a small instance of the idea.

in-toto and SLSA. in-toto binds signed statements about software artifacts to subjects identified by digest (Torres-Arias et al., 2019; in-toto project, 2026), and SLSA defines levels of build integrity and a provenance predicate (SLSA, 2026). Where they are ahead: they cover how an artifact was built and by whom, across a supply chain and several parties, which Sabline says nothing about. Sabline's documents describe what a source text declares, not how or by whom an artifact was built. sabline-spec defines an in-toto predicate type for the audit, and sabline-lang 4.2.0 writes statements of it; its release workflow signs one for an example program with Sigstore's tools and verifies it.

SARIF. SARIF is a common format for the findings of static analysis tools (OASIS, 2020). Where it is ahead: it is an OASIS standard that code-scanning tools and their hosts already read, where sabline-spec's formats are one project's. sabline-lang writes its compile errors, unproven promises and capability widenings as SARIF for code scanning; the audit itself describes a program rather than listing findings, and is not SARIF.

Gradual attacks. Hills, Caspary and Cooper Stickland show that a coding agent can spread a covert side task across pull requests so that no single diff looks decisive, that such attacks evade diff monitors, and that a stateful monitor tracking buildup across pull requests detects them better (Hills et al., 2026). Where it is ahead: it measures a threat and a defence against it, and its monitor sees buildup of any kind, where the ratchet sees one kind. The ratchet is not a monitor of intent. It is a deterministic comparison of what a repository's Sabline programs need with a baseline the repository declared, at every change. It sees the part of a gradual attack that needs capability the baseline did not declare, and nothing else: a side task that stays within the declared surface, or is not written in Sabline, is outside it.

AI-first languages. A catalogue of languages designed for models rather than for people lists 41 projects in three camps: syntactic ones that strip ambiguity at the token level, verification ones that make contracts mechanically checkable, and orchestration ones that treat the matter as agent coordination (Allan, 2026a). Sabline belongs in the second and third, and several entries occupy close ground, reached independently: sabline-lang cites none of them. Boruna is a deterministic, capability-safe workflow language whose functions declare effects in a `!{...}` clause, whose virtual machine gates every call against an operator's policy over capabilities, endpoints, models and per-step budgets, and whose runs produce hash-chained evidence bundles that replay to identical outputs (escapeboy, 2026); it parses `ensures` without enforcing it and has no prover, its evidence describes a run where Sabline's audit describes a source text and binds to in-toto and SARIF, and it grants nothing by default, where Sabline without a budget granted all seven effects until version 5.0 and grants `io` - the console alone - from it. Boruna's default is still the stricter of the two: it grants nothing, and a Sabline program with no budget can print. Thermite mandates `req`, `ens` and `fx` on every function and settles each obligation separately on a five-rung ladder from reconstruction in Lean down to an always-active runtime check, recording engine and level per clause (dollspace-gay and Levesque, 2026); its verification is well ahead of Sabline's single Z3 tier, and the difference is who writes the policy - Thermite derives a seccomp filter from the program's own `fx`, where a Sabline budget is the operator's and may be narrower than the

program declares. Vera makes `requires`, `ensures` and `effects` mandatory and sorts each obligation into Z3's decidable fragment or a compiled runtime guard, with a conformance corpus and a draft specification (Allan, 2026b), but bounds no paths or hosts at run time and keeps no audit record. AILANG, the closest of them, has a paragraph of its own below. No entry combines all four of effects in signatures, an operator's budget refused against at the operation, a prover with a runtime fallback, and a published format for the declared surface.

AILANG. AILANG is a purely functional, effect-typed language for code a model writes, in which a function's effects are rows in its type and a run is granted capabilities at the command line (Edmondson, 2026). Read in full at version 0.42.0, its documentation describes more than the category grant this paper's earlier versions credited it with. Where it is ahead: labels for information flow with an open vocabulary - `<pii>`, `<email>`, a tenant's name - with a sink refusing a label and declassification an effect of its own, where Sabline has the one `Secret of T`; more effects, among them a process effect whose allowlist narrows to a subcommand (`git:status`) and an AI effect across providers, where Sabline's `ffi:` grants a whole module and it has no model of a subprocess; counts in the signature, per function and with a minimum, beside an operator's per-run ceiling in its policy mode; effect ceilings per package, checked by the compiler and by its registry; a Z3 fragment covering strings, lists, records and bounded recursion, with a refuted contract blocking a package's publication; property-based tests; and a far larger public evaluation of how well models write it, against a Python baseline, losses included. Where they differ: its finer grants - a domain allowlist, one filesystem root, a process allowlist - belong to the invocation rather than the type, as Sabline's do, but a refused host or command comes back as an error value the program can handle, where a Sabline refusal ends the run; its documentation describes no confinement by the operating system beneath the interpreter, leaving CPU, memory and host isolation to a container, and no report of what a program can touch before it runs; its own audit found and fixed escapes from both its filesystem root and its domain allowlist in version 0.41.0 (21 September 2026); and checking contracts at run time is a flag rather than a fallback, with effectful functions outside what its prover takes.

6. Limitations

These are stated as sabline-lang's THREAT_MODEL.md states them.

- **Only code written in Sabline.** A model asked for Sabline may hand back Python. Nothing here applies to code the Sabline runtime does not run.
- **A granted `ffi` module.** A granted module can do whatever it can do. `ffi:os` is the operating system; `ffi:subprocess` is a shell. The grant narrows which modules a call may reach, not what a module does.
- **Logic errors with no contract.** A function that returns the wrong number and promises nothing is correct as far as the compiler knows. Benchmark program `04c` is this, and is a miss.
- **The meaning of text.** A program that prints a shell command for its caller touches nothing. No effect system can tell it from a program that prints the same words as a warning, and Sabline does not try.
- **One maintainer.** The project is maintained by one person. Fixes to soundness and sandbox reports are promised within a week, and nothing else is promised.
- **Not a security boundary.** The budget is enforced by a Python interpreter in the same process as the compiler, and an altered compiler holds nothing. Until 8.4.0 a defective one held nothing either. From 8.4.0 the operating system holds the same budget under the interpreter, and a test on every platform has the runtime itself attempt a read, a write, a connection, a new process and a signal outside the budget: on Linux the kernel refuses all five; on macOS reads are refused only under the home directory; on Windows a new process is refused, a write is refused only under a budget that grants none, and reads and the network are not held at all. With the interpreter's own budget checks removed, 19 of the 39 applicable escape attempts of the suites still fail on Linux, at the kernel, and 8 of 38 on Windows; the rest - a host inside a `net` grant, the operation counts, `env`, and a Python object reached inside the same process - are held by the language alone. A granted `ffi` module widens what the system is asked to hold, to nothing for `ffi:os`. Side channels, resource use below the limits, where a granted host name

resolves, and what a granted host does with a request are outside the model.

- **The ratchet.** The declared surface cannot widen without the check failing (section 2.4); these are its known limits, each held as a case in `check_ratchet.py` and in the corpus with the outcome it has today. Two widenings that pass: a function renamed in the same change that gives it an effect its program already had evades function-level attribution (W5), though the declared surface did not widen; and what a granted `ffi` module does is outside the program's text. Three changes that do not widen and are reported as widenings, each a conservative false positive: a loop limit behind a function call (`for i in 0 to limit()`) loses its bound and is reported as unbounded; a path passed to a helper as a parameter is taken as unscoped even when every caller passes a literal; and a path written with a backslash is covered only by the same text, so `data\in.csv` is outside a recorded `data` even on Windows, where it is inside. The ratchet also guards nothing if the check is not required, and an edit to the baseline is an accepted widening whose review is the control.
- **The evaluation.** Section 4.1's benchmark was written by this project and is small, and section 4.2's translations of it were written by AI agents for this project; no user study has been done; the ratchet's properties rest on its definition and its cases, not on a measured detection rate.
- **An evaluation against an adversary, without the adversary's model.** Section 4.3 runs Sabline on AgentDojo's Slack suite with AgentDojo's reference solutions in place of a model: 21 of 21 tasks completed under every budget, and of 105 attacks 19 landed under a task budget and 9 under that budget with each tool's arguments pinned to what the user task's own text names. The attack figures are upper bounds on what those budgets let through, not measured rates at which a model is steered, and nothing here measures a model writing Sabline under attack. The 9 that remain are laundering with an argument no operator could have written down, which a budget over effects and tools cannot see: Sabline has no per-value provenance, and CaMeL and ChainCaps, which have a number with a model in the loop, should do better on them. Every one of the 19 that landed had the attacker's value written into the program rather than read at run time, which is an artefact of a harness with no model in it, and the two injection tasks whose payload really is read at run time never reach any task budget's door at all. It is one suite at one version, run by the project it measures, which is the first of the three weaknesses Abdelnabi and colleagues name (Abdelnabi et al., 2026).

7. Conclusion

Sabline puts what a function may do in its signature and checks it across the call graph; refuses, while a program runs, every operation outside a budget its operator wrote, in a way the program cannot catch; proves contracts where the prover can; and holds a repository's declared capability surface to a baseline that only an edit can widen. The first three can be tested program by program, and on the first eleven categories and the thirteenth to fifteenth of a 102-program benchmark they caught 61 of 63 defects; the fourth's comparison, applied to two versions of a dependency, caught the three defects of the twelfth; and no correct program of those categories was flagged. Five categories written for it to lose did what they were written for: it missed eight of their fourteen defects, stopped four of their twelve correct programs, and five other tools each did better than it somewhere. The fourth is a property, not a rate: under a required check, the declared surface does not widen without the check failing, however the change is divided among commits. What it does not do - attribute an effect to the right function across a rename, see into a granted module, tell harmless text from a dangerous command - is stated here as precisely as what it does, because the second list is only worth believing if the first one is.

Use of generative AI

The software described here was developed by the author with the assistance of AI coding agents, and the first draft of this paper was written by an AI coding agent from the repository's contents under the author's direction. The author specified the structure and claims, reviewed and revised the text, and is responsible for its accuracy. Every number reported was verified against the repository files named in the reproducibility section. Three adversarial reviews by AI systems, credited in the repository's HALL_OF_FAME.md, found defects that are recorded in its changelog.

Reproducibility

This paper describes Sabline 4.2.1 and sabline-spec 0.5.1, and every number in it was verified against those two tags except the benchmark's and the confinement figures of section 6: the benchmark figures of the abstract, sections 4.1 and 4.2, Tables 1 and 2 and the conclusion are from `benchmark/results.json` and `benchmark/competitors/results.json` at the commit that added the benchmark's categories 16 to 20 (pull request #105; 8.7, not released), section 4.3's are from `evals/agentdojo/results.json` at the commit that added it (pull request #103), and section 6's counts of escape attempts stopped by the kernel are from `tests/confine/kernel-linux.json` and `kernel-windows.json` at the same tag. Releases after 4.2.1 postdate the paper and are not reflected in it: 4.3.0 added `Money of CUR`, an exact decimal whose split is proven to add back up; 4.3.1 made a proof that exhausts its time budget say so rather than fall silently back to a runtime check; 4.4.0 added a reference platform service; and 5.0.0 made `io` the budget a run gets when nobody writes one, where every version this paper measured granted all seven effects - the related-work paragraphs on WASI and on Boruna say what changed, and sabline-spec 0.6.0 restates its sections 4.4 and 4.6 to match; 6.0.0 and 7.0.0 added `Secret of T`, a value a program cannot print, send or branch on without declassifying it with a stated reason; 7.1.0 added `sabline deps-diff` and the benchmark's twelfth category; 7.1.1 made 7.1.0 import on Python 3.10 and 3.11, and changed nothing the benchmark measures; 7.1.2 and 8.1.1 closed two ways for a planted proof cache to report a false promise as proven; 7.2.0 changed how releases are made; 8.0.0 made four changes that break programs or commands 7.x accepted, and added the benchmark's thirteenth category; 8.1.0 added receipts of a run; 8.2.0 split the compiler into one module per stage and keeps no proof cache at all; 8.3.0 added an evaluation profile, a comparison and a replay of receipts, and the benchmark's fourteenth and fifteenth categories; 8.4.0 asks the operating system to hold a run's budget, which sections 3 and 6 now describe; 8.5.0 added a tool door for a runner and the batteries built on it; and 8.6.0 renamed the project from Velaris to Sabline, which is the name used throughout here. Later patches corrected documentation and packaging. What each one changed is in the two repositories' changelogs. The tags below are therefore the ones to check out, not the current releases.

Four files that section 5 and section 6 name are plans and decisions rather than measurements, and they are at the tip of `main` rather than at either tag: `decisions/0004-untrusted.md`, the design for a mark on untrusted input, whose amendment takes ChainCaps' propagation rule; `plan/9.0.md`, the milestones of the Rust runtime; `plan/8.7.md`, the plan for the evaluations of sections 4.2 and 4.3 and for those not yet run; and `docs/runner.md`, which records the laundering case 0004 answers. None of the four has shipped, and nothing in this paper is measured from any of them. The 9.0 pre-releases published so far - `9.0.0-alpha.1` to `alpha.4` - publish the `sabline-rt` crate alone and change nothing the paper measures.

The numbers in this paper can be regenerated from the two repositories at their tags:

```
git clone https://github.com/gowrishankar-infra/sabline-lang
git clone https://github.com/gowrishankar-infra/sabline-spec
git -C sabline-spec checkout v0.5.1
cd sabline-lang

# Table 1, at v8.4.0 (Deno 2.x on PATH for the Deno column; 4 to 8 minutes)
git checkout v8.4.0
pip install ".[full,test]"         # the prover, the native compiler, jsonschema
python benchmark/run.py --check    # exit 1 if any verdict differs

# section 6's confinement counts, on the system they are for
python check_confine.py            # fails if a recorded kernel stop gets through

# everything else, from v4.2.1
git checkout v4.2.1
pip install ".[full,test]"

# Figure 1a
sabline audit examples/effects.vel
```

```
sabline examples/effects.vel --allow io,clock,rand,fs:read:./data

# sections 4.4 and 4.5
python check_sandbox.py
python check_library.py
python check_ratchet.py
python check_refusals.py
python check_fallible.py
python check_termination.py
sabline conformance --corpus ../sabline-spec/tests
python build_conformance.py --check ../sabline-spec/tests
sabline capabilities check .

# the same without the prover
python -m venv bare
bare/bin/pip install ".[test]"     # bare\Scripts\pip on Windows
bare/bin/python check_sandbox.py   # and each command above
```

Where each number comes from:

Number	File
102 programs, 80 dangerous, 22 controls, twenty categories; 54/16/10, 8/43/29, 0/32/48; 4, 0 and 0 false positives; seven Sabline catches with the task broken; Sabline 8.6.0, Deno 2.9.6, Python 3.13.13, Windows 11; 5 s timeout, 256 MB cap; the misses `04c`, `09c` and eight in categories 16 to 20; 61 of 63 in categories 1 to 11 and 13 to 15; category 12's three upgrades and one control; category 13's program, its verdicts and E318; categories 14 to 20; 08f's catch moved to while running	`benchmark/results.json` at the commit that added categories 16 to 20
Table 2 and section 4.2: the five competitors, their versions, grants and verdicts; 18 programs where one does better; 12 programs in CaMeL's scope; 19d stopped by none	`benchmark/competitors/results.json` at the same commit, and `docs/competitors.md` built from it
Section 4.3: AgentDojo 0.1.35, Slack suite v1; 21 user tasks, 5 injection tasks, 105 attacks; utility 21 of 21 under all three budgets; 105 attacks land under `--allow all`, 19 under the task budget (12 visiting a web page and 7 sending a message, 86 refused with E321) and 9 under the task budget with arguments pinned; all 19 have the attacker's value written into the program, none read at run time; 32 of 80 arguments pinned, 18 of 21 tasks with an argument that cannot be known in advance	`evals/agentdojo/results.json`, re-derived by `agentdojo_eval.py --check`
the same table at 8.4.0: 76 programs, 66 dangerous, 10 controls, fifteen categories; 52/12/2, 8/34/24, 0/31/35; 0 false positives	`benchmark/results.json` at v8.4.0
three of each six programs written against the tools; about four minutes for a full run, five to eight on a slower machine	`benchmark/README.md` at v8.4.0

Number	File
the 68 programs of categories 1 to 13 unchanged in verdict since 8.2.0; ten identical runs at 8.3.0; categories 14 and 15 added in 8.3; every verdict unchanged at 8.4.0	`benchmark/results.json` at v8.2.0, v8.3.0 and v8.4.0, and `CHANGELOG.md`, 8.3 and 8.4 entries
the runtime's own five attempts and what each system refuses; 19 of 39 and 8 of 38 escape attempts stopped at the kernel with the budget checks removed	`check_confine.py`, and `tests/confine/kernel-linux.json` and `kernel-windows.json`, at v8.4.0
the 67 programs of categories 1 to 12 unchanged in verdict and evidence since 7.1.1	`benchmark/results.json` at v7.1.1 and at v8.2.0
category 13 added in 8.0	`CHANGELOG.md`, 8.0 entry
ten identical runs at 7.1.0; the verdicts of the 63 earlier programs unchanged from 4.1.0; the run at 7.1.1 differing only in its version	`CHANGELOG.md`, 7.1 and 7.1.1 entries
the same table from Sabline 3.0.0	`benchmark/RESULTS.md` at v4.0.0, and `CHANGELOG.md`, 4.1 entry
seven effects; the refusal codes E310, E311, E313, E314, E315	`SPEC.md` section 7 and 7.1
E407, E520, E700, E701, E705, E706; 62 error codes	`sabline.py`, `ERROR_TABLE`
Figure 1a: the program, its effects and paths, line 13	`examples/effects.vel`
13,497 lines	`sabline.py`
34 escape attempts and 16 honest programs	`check_sandbox.py`, `ESCAPES` and `HONEST`
Linux, Windows and macOS; Python 3.10 and 3.12; with and without the prover	`.github/workflows/test.yml`
114, 196, 21, 26, 44 checks; 193 and 11 without the prover	`CHANGELOG.md`, 4.2 entry
456 cases: 307, 40, 109; 280 budgets, 27 audits; 19 left out	sabline-spec `tests/index.json`
the six-commit history; line 2 of `lib/deliver.vel`; 10 to 1000, 640 to 1000, and Figure 1b	`check_ratchet.py`, `SEQUENCES`
five known limits	`check_ratchet.py`, `CHECKS`; `CHANGELOG.md`, 4.0 entry
172 programs, 22 not compiling	`sabline.capabilities`

References

Abdelnabi, S., Hicks, C., Rieck, K., and Sadeghi, A.-R. (2026). Measuring security without fooling ourselves: Why benchmarking agents is hard. arXiv:2605.22568.

Allan, A. (2026a). Agent languages: a catalogue of programming languages designed for AI agents. https://agentlanguages.dev. Read at the version of 11 September 2026, tracking 41 projects. Mirrored at https://research.tedneward.com/places/agentlanguages.html.

Allan, A. (2026b). Vera: a language with mandatory contracts and tiered verification. Version 0.1.13. https://github.com/aallan/vera.

Bühler, C., Biagiola, M., Di Grazia, L., and Salvaneschi, G. (2026). AgentBound: Securing execution boundaries of AI agents. *Proceedings of the ACM on Software Engineering*, 3(FSE):2141–2164. Article FSE096, FSE 2026. arXiv:2510.21236.

Bytecode Alliance (2026). Filesystem sandbox escape when paths or symlinks contain trailing slashes. RUSTSEC-2026-0269; GHSA-vqjp-4c8c-hfgg. Published 2026-08-20; CVSS 8.8. Patched in wasmtime 24.0.13, 36.0.14, 46.0.3 and 47.0.4. https://rustsec.org/advisories/RUSTSEC-2026-0269.html. Accessed 2026-09-22.

de Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer.

Debenedetti, E., Shumailov, I., Fan, T., Hayes, J., Carlini, N., Fabian, D., Kern, C., Shi, C., Terzis, A., and Tramèr, F. (2025). Defeating prompt injections by design. arXiv:2503.18813.

Debenedetti, E., Zhang, J., Balunović, M., Beurer-Kellner, L., Fischer, M., and Tramèr, F. (2024). AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. arXiv:2406.13352.

Dennis, J. B. and Van Horn, E. C. (1966). Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155.

Deno (2026). Security and permissions. https://docs.deno.com/runtime/fundamentals/security/. Accessed 2026-09-11.

dollspace-gay and Levesque, M. (2026). Thermite: contract-first verification with per-obligation assurance levels. https://github.com/dollspace-gay/Thermite.

E2B (2026). Security and compliance. https://e2b.dev/security. Accessed 2026-09-22.

Edmondson, M. (2026). AILANG: an effect-typed language for AI-generated code. Version 0.42.0, 23 September 2026; documentation at https://ailang.sunholo.com/, read in full on 25 September 2026.

escapeboy (2026). Boruna: a deterministic, capability-safe programming language and framework for LLM-native applications. Version 3.2.0. https://github.com/escapeboy/boruna.

Hills, J., Caspary, I., and Cooper Stickland, A. (2026). Distributed attacks in persistent-state AI control. arXiv:2607.02514.

in-toto project (2026). in-toto Attestation Framework. https://github.com/in-toto/attestation. Accessed 2026-09-11.

Jiang, X., Yang, S., Li, Z., Liu, L., Yu, H., and Liu, Y. (2026). ChainCaps: Composition-safe tool-using agents via monotonic capability attenuation. arXiv:2605.26542. Second Workshop on Agents in the Wild: Safety, Security, and Beyond (AIWILD) at ICML 2026.

Lucassen, J. M. and Gifford, D. K. (1988). Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*, pages 47–57.

Microsoft (2026). Wassette: a security-oriented runtime that runs WebAssembly components via MCP. Version 0.7.1, 12 September 2026. Documentation: https://microsoft.github.io/wassette/. Accessed 2026-09-22.

Miller, M. S. (2006). *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University.

Modal (2026). Security and privacy at Modal. https://modal.com/docs/guide/security. Accessed 2026-09-22.

OASIS (2020). Static Analysis Results Interchange Format (SARIF) Version 2.1.0. OASIS Standard.

Odersky, M., Zhao, Y., Xu, Y., Bračevac, O., and Pham, C. N. (2026). Securing agents with tracked capabilities. In *Proceedings of the ACM Conference on AI and Agentic Systems (CAIS '26)*, pages 812–838. ACM. Preprint as "Tracking Capabilities for Safer Agents", arXiv:2603.00991. Implementation: https://github.com/lampepfl/tacit.

Palakurthi, G. S. (2026a). Sabline. Version 4.2.1. https://github.com/gowrishankar-infra/sabline-lang.

Palakurthi, G. S. (2026b). sabline-spec: the Sabline capability format. Version 0.5.1. https://github.com/gowrishankar-infra/sabline-spec.

SLSA (2026). Supply-chain Levels for Software Artifacts. https://slsa.dev. Including its provenance predicate. Accessed 2026-09-11.

Tan, H., Wang, Y., Zhang, P., Li, S., and Shen, J. (2026). ETAS: An effect-typed language for agent systems. arXiv:2607.17780. Version 1, 20 July 2026.

Torres-Arias, S., Afzali, H., Kuppusamy, T. K., Curtmola, R., and Cappos, J. (2019). in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium*, pages 1393–1410.

WebAssembly Community Group (2026). WASI: the WebAssembly System Interface. https://wasi.dev. Releases 0.1, 0.2 and 0.3, also called Previews 1 to 3. Accessed 2026-09-11.

Zhou, T., D'Antoni, L., and Polikarpova, N. (2026). Language-based agent control. arXiv:2605.12863. Prototype: TypeGuard, in Haskell.