Verifying Fault Robustness by Type Checking: A Verified Information-Flow Approach*

Vadim Zaliva[1][0000−0002−9145−3288], Valerii Huhnin[2][0009−0006−0572−7809], and Jeffrey S. Foster[1][0000−0001−8043−1166]

[1] Tufts University, Medford, MA, USA
{vadim.zaliva,jeffrey.foster}@tufts.edu
[2] Université Paris Cité, Paris, France
valerii.huhnin@etu.u-paris.fr

Abstract. In many existing software systems, developers must handle errors by checking for sentinel return values indicating a fault. It is all too easy to accidentally omit such checks, which can lead to misbehaviours when faults occur. Inspired by concepts from secure information flow, in this paper we take a novel approach to building a verifier that catches such omissions. Our approach is based on the observation that values that arise from faults should not flow to observable program outputs. More specifically, we introduce *fault robustness*, a novel 2-safety property that holds if, under any two states in which only the faulty inputs differ, a given expression always produces the same observable output. Finally, we develop an executable typechecker that serves as a certified fault-robustness verifier by tracking the flow (including indirect flows) of faulty, or *poisoned*, values through a program to ensure that observable outputs remain *clean*. Additionally, the type system tracks the program's own checks of sentinel values and marks a checked value, together with the values correlated with it, as clean in the branch where the check succeeds. We formalise our type system and fault robustness in Lean 4 and prove that if potentially faulty inputs are initially typed as poisoned and the program typechecks with observable outputs typed clean, then the program satisfies fault robustness.

Keywords: fault robustness · fault isolation · information-flow analysis · flow-sensitive typing · type systems · operational semantics · formal verification · Lean

1 Introduction

Among the various paradigms for error handling, the use of sentinel return values is a prevalent idiom [45], especially in system-level programming languages such as C and C++, and in operating system APIs like POSIX. The core idea is that a function or system call returns either the actual result of the operation or a special value (e.g., `NULL` or a negative integer) that signals failure. Even

* Preprint. Accepted at VMCAI 2027; the published version may differ.

modern programming languages often still use sentinel return values in their APIs or standard libraries (e.g., `InputStream.read()` in Java, `indexOf()` in JavaScript), and the validity of one result may depend on another (the output parameter of C#'s `TryParse()` is valid only if it returns `true`).

The responsibility for checking and handling these return values falls to the caller. However, in practice, this checking is frequently omitted, or the error value is propagated through function calls or stored in variables and memory, making it difficult to guarantee that it is checked along all paths [22,45,24,15,25]. Moreover, while this style of error reporting has mostly been superseded by other approaches such as exceptions, optional types, and effect systems, it remains common: programmers may still choose, or be required, to use sentinel return values, especially when interfacing with system APIs or libraries that expose traditional error-reporting conventions, and many large, often mission-critical legacy codebases continue to rely on this pattern.

In this paper, we introduce a new approach to reason about whether a program handles sentinel return values correctly. Our key insight is to view correct sentinel handling as an information flow problem: If a sentinel indicates a fault has occurred, then any potentially faulty values should not flow to observable program outputs, as an output computed from a faulty value is unreliable. Moreover, we aim to reason about programs in which whether inputs are faulty depends on a predicate that might be tested far from the source of the input. We introduce a novel type system that carries out such reasoning on a core language and formally develop a notion of *fault robustness*: informally, the observable outputs of a program do not depend on inputs that turn out to be faulty.

We formulate our type system, inspired by secure information flow systems [44], on the *Poison* language, an imperative core calculus in which external values, including functions, can be given types with *poison*, meaning they are potentially faulty. More specifically, external values are given *qualified types* of the form $T \triangleright p\,\{P\}$, where T is the underlying type, p is their poison status (either poisoned or clean), and P is a *guard*: it carries a tag k and, optionally, a pure expression that, if true, indicates the value can be de-poisoned, i.e., treated as clean after the predicate is tested at run time. Each tag may be associated with multiple values. When they carry guard predicates, those predicates must match exactly, and a single check de-poisons every value sharing the tag.

Our main result (Section 4.2) is that type checking enforces fault robustness: if potentially faulty inputs are typed as poisoned and the program typechecks with observable outputs typed clean, then the program satisfies fault robustness. We prove this theorem in Lean 4 (Section 5), mechanising the operational semantics, typechecker, and fault-robustness property. In analysis terms, the type system is a flow- and context-sensitive analysis whose soundness proof is machine-checked and stated about the executable typechecker; Section 7 relates it to abstract interpretation. The optional Appendix A, intended to be read after the main paper, gives additional insight into the design of the type system and the details of the Lean mechanisation.

In summary, this paper makes the following contributions:

- **Information-flow analysis for fault robustness.** We replace the classical *high/low* security lattice with a *non-faulty/potentially faulty* distinction and use it to track fault propagation through data dependencies and control flow (Section 2).
- **Type system for fault propagation.** Unlike classical information-flow analyses, where labels can only move up the lattice, our system allows them to move down when a guard predicate is matched, modelling the program's own fault checks. Moreover, in our system a single predicate check can recover multiple correlated values, including heap-resident state, through a shared tag, directly modelling real-world patterns such as "return code plus out-parameter" (Section 3).
- **Formal fault-robustness property.** We formulate a 2-safety property [7] that extends standard noninterference: faulty critical inputs may differ between runs, while observable outputs must agree (Section 4).
- **Certified checker.** We develop an executable typechecker and prove in Lean 4 that it conservatively enforces the fault-robustness property, so that it serves as a certified fault-robustness verifier (Sections 5 and 6).

2 Modelling Fault Propagation and Handling via Information-flow Analysis

Unchecked return values have repeatedly led to serious flaws and security vulnerabilities. Recorded CVEs include privilege escalation in `dmcrypt-get-device` [9] and `doas` [10], authorisation bypass in Apache Accumulo [12], disclosure of process memory in Google Chrome [11], denial of service in CUPS [8] and `libarchive` [14], and requests processed after a failed initialisation in the Linux Tegra crypto driver [13].

Studies show that unhandled error returns are pervasive. A static analysis study of Linux file systems and storage drivers [22] found that 1153 of 9022 calls that may return an error code, about 13%, did not handle it properly, more often in complex, layered code and on write paths (19.6% of write errors against 4.3% of read errors); some functions' return values were consistently ignored, suggesting developers assumed those calls "cannot fail". Another study uncovered "hundreds of confirmed error-handling bugs" in dozens of Linux file systems and drivers [20], which could cause crashes, memory leaks, and silent data loss, and named *incorrect error code propagation* as one root cause: a low-level error returned from a device driver is lost as it propagates across several levels of abstraction, one of the central problems we address. More broadly, a 2021 study [47] found that at least 34% of the latest 100 CVE-assigned vulnerabilities in the Linux kernel involve incorrect error-handling logic.

Motivating Example. To illustrate how our system addresses this situation, we consider a real-world bug in the Linux kernel's LP5523 LED driver that was caused by ignoring the return value of `lp55xx_read` [34]. The code fragment in

```
let _ = lp55xx_read status in
  (status := !status & ENG_STATUS_MASK;
   if !status ≠ ENG_STATUS_MASK then
     (eng := NOT_READY; ERROR)
   else
     (eng := READY; OK))
```

Listing 1. Motivating example in Poison (buggy variant).

Listing 1 is a Poison model of the original driver's C code, using the language described in Section 3.

This fragment begins with a call to `lp55xx_read`, which writes to its parameter `status` and returns a non-negative code on success or a negative code on failure. The code, however, discards this return value (binding it to `_`) and never checks it, and thus `status` is untrustworthy. Therefore the next line, which combines `status` with a mask, yields a result that is also untrustworthy, as it is derived from a potentially faulty value. Then in the next line, using this value as the condition of an `if` statement makes both the engine-state cell `eng` and the code it returns (`OK` or `ERROR`) untrustworthy.

Fault Propagation via Information-flow Analysis. Secure information flow type systems [44] classically treat program inputs and outputs as either *high* or *low* security, and they enforce the property that high-security inputs may not affect low-security outputs, through both direct and implicit flows. The key observation behind our approach is that we can use similar ideas to prevent untrustworthy inputs from affecting observable outputs. More specifically, we introduce a type system that distinguishes between *clean* inputs, which are guaranteed non-faulty, and *poisoned* inputs, which may be faulty. As with high and low, clean and poisoned form a two-point lattice with poisoned being the top of the lattice since a clean value can flow to where a poisoned value is expected. Our type system then guarantees that poisoned inputs do not affect observable, clean outputs.

For example, in the code in Listing 1, we can annotate the call to `lp55xx_read` as poisoning `status` (poison enters only through external functions and values). The update to `status` with the bitwise mask preserves its poison. Then the branch on `status` creates an implicit flow (essentially, the program counter becomes poisoned inside the branches of the conditional), and therefore the write to `eng` and the returned `OK`/`ERROR` result both become poisoned. The type system therefore warns that both observable outputs—`eng` and the returned code—are poisoned and cannot be trusted.

In this example we model the return value of `lp55xx_read` as a sentinel linked to a mutable location `status`. A guard predicate partitions the sentinel's values into successful *actual values*, which the program may use, and one class of error values, which are not supposed to be used and should not influence the outputs. Under this model, the type system does not distinguish error values and considers only this success/failure partition. This is the right fit for ensuring that the program always checks for possible failures correctly and does not mistake sentinel error values for legitimate data. It does not support a scenario in which

a program must return different results for different failure codes: our certified property is that observable outputs are independent of faulty values, and it does not support controlled dependence on them (Section 4). This could be viewed as a limitation, but specifying which such dependencies are permitted would require substantially more machinery, so we pragmatically leave it for future work.

Predicate-Based De-Poisoning. Handling faulty return values requires reasoning about the program's own validity checks: unlike traditional secure information-flow analysis, we therefore allow the poison status of a value to change non-monotonically, meaning that a poisoned value can be made clean again. This *de-poisoning* happens when the program checks a value to detect a fault and that check matches the guard predicate associated with the value.

In our system, a poisoned type can carry a guard describing when the value may be considered clean. When a conditional tests exactly that guard's predicate, the type system de-poisons the value in the successful branch. For example, if `ret` is a poisoned integer guarded by the predicate $x \geq 0$, then in the expression `if ret ≥ 0 then ret else 0` the type system recognises that the condition matches the guard, so `ret` is treated as clean in the `then` branch; in the `else` branch the literal `0` is unconditionally clean. Either way, the whole expression is clean. By contrast, `if ret ≥ 0 then ret else (0 - ret)` is typed *poisoned* when the check fails: the `else` branch returns a value computed from the unvalidated `ret`, so the output may still depend on the faulty value.

Listing 2 revisits our motivating example, now showing the full annotation syntax and including a fix that checks the return value of `lp55xx_read`. Throughout the listing, annotations belonging to the type-system and verification layer, all written by the programmer, are shown in grey, while the runtime language, including the `extern` declarations that bind host-provided values, is in black. Identifiers such as `@234` link each extern to its host implementation (see the accompanying development). The type annotations for externs use *qualified types* `(ETyp ▷ poison {tag, x⇒pred})`, where `ETyp` is the base type and `poison` is `clean` or `poisoned` (i.e. possibly faulty), and the braced part is an optional guard with a numeric tag and a predicate over formal parameter `x`. The declaration of `eng` introduces a clean external *reference* to a poisoned integer memory cell. By contrast, `status` is an ordinary local reference to a memory cell initialised to 0 and not poisoned. The remaining declarations introduce external integer constants `ENG_STATUS_MASK`, `NOT_READY`, `READY`, `ERROR`, and `OK` that are unconditionally clean. `lp55xx_read` then takes a clean integer reference `r` and returns a possibly-poisoned value guarded by the predicate $x \geq 0$ and tag 100. The annotation also includes an `effects` keyword that marks the cell its argument `r` points to with the same tag 100, but no predicate. Because the return value and the written cell share tag 100, a single validation check can restore the cleanliness of both: if the tagged condition $x \geq 0$ is known to hold, the type system may treat all values associated with tag 100 as clean. The literal 100 is a tag *index*, local to the declaration: each application of an extern function allocates fresh tags for the indices in its signature, so the results of different calls

```
1  let status = ref 0 in
2  let eng = extern ref @234 : (ref int ▷ clean {}), points_to: (int ▷ poisoned {}) in
3  let ENG_STATUS_MASK = extern val @235 : (int ▷ clean {}) in
4  let NOT_READY = extern val @236 : (int ▷ clean {}) in
5  let READY = extern val @237 : (int ▷ clean {}) in
6  let ERROR = extern val @239 : (int ▷ clean {}) in
7  let OK = extern val @240 : (int ▷ clean {}) in
8
9  let lp55xx_read =
10   extern fn @238 r : (ref int ▷ clean {}) → ((int ▷ poisoned { 100, x ⇒ (x ≥ 0) })),
11     effects: { !r ▷ (int ▷ poisoned { 100 }) }
12 in
13
14 let ret = lp55xx_read status in
15 if ret ≥ 0 then
16   (status := !status & ENG_STATUS_MASK;
17    if !status ≠ ENG_STATUS_MASK then
18      (eng := NOT_READY; ERROR)
19    else
20      (eng := READY; OK))
21 else
22   (eng := NOT_READY; ERROR)
```

Listing 2. Predicate-based de-poisoning: the returned code and engine-state cell stay clean.

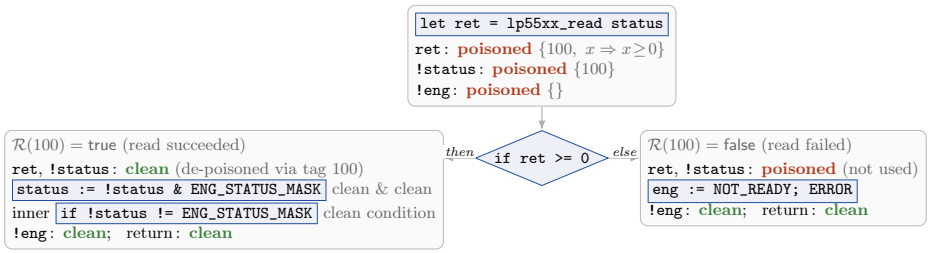


Fig. 1. Typing walkthrough of Listing 2.

never share a tag and checking one does not de-poison another. For the single call in Listing 2 we write 100 for the tag itself.

Figure 1 traces how the inferred types evolve. After the call to `lp55xx_read` (line 14), both `ret` and `!status` are poisoned and carry tag 100, and the check `ret ≥ 0` (line 15) is recognised as the validation test associated with that tag. Its outcome is fixed by the *regime* (Section 3.2), an assumption about which tagged inputs are faulty, so each regime selects one branch and only two cases remain. In the successful-read case, $\mathcal{R}(100) = \mathsf{true}$, the analysis treats every value tagged 100 as clean in the `then` branch, so the masking of `status`, the writes to `eng`, and the returned code (`OK` or `ERROR`) are all clean. In the failed-read case, $\mathcal{R}(100) = \mathsf{false}$, the `else` branch (line 22) marks the engine as stopped, writing the clean constant `NOT_READY`, and returns the clean constant `ERROR`. Either way, both observable outputs (the returned code and the engine-state cell) have clean types.

Thus, our type system guarantees that even if the sensor reading is faulty, that faulty value is isolated and will not affect observable program outputs.

```
let a = ref 0 in
let x = extern val
  @250 : (int ▷ poisoned
    { 1, v ⇒ (v > !a) }) in
if x > !a then
  (a := 100; x)
else 0
```

(a)

```
let r1 = ref 0 in
let r2 = ref 2 in
let r =
  if flag then r1
          else r2 in
(r := poisoned_value;
 !r2)
```

(b)

```
let x = 1 in
let a = ref 0 in
let f =
  fn y : int ⇒
    x + y + !a : int in
let u = f 1 in
(a := poisoned_value; f 1)
```

(c)

Fig. 2. Examples of fault-handling patterns that shaped the type system. Here `flag` stands for a clean boolean and `poisoned_value` for a poisoned input.

Notice the subtle difference between this property and standard information flow. In the latter, we wish to ensure that no high-security input affects a low-security output. But in our fault propagation setting, we want to ensure that if an input is determined to be faulty, then it will not affect an observable output; yet an input at the same program position that is not faulty may affect the program output.

Challenges in Modelling Poison Propagation. Modelling this reasoning via a type system requires tracking how poison propagates through every construct in the language, and recognising when a conditional test matches a guard predicate so that de-poisoning can be applied (Section 3.3). Figure 2 distils three of the many challenges this raises into small fragments whose final expressions must all be typed poisoned, even though each looks harmless.

In Figure 2(a), the test `x > !a` de-poisons `x`, but the guard predicate itself reads the mutable cell `a`, and the write `a := 100` invalidates the earlier check. To ensure de-poisoning is tracked correctly under mutation, the type system records the locations each satisfied guard depends on and revokes the guard when any of them is written, so `x` reverts to poisoned (Section 3.4). In (b), `r` may alias `r1` or `r2`, so, conservatively, the assignment through `r` must poison both cells. Reference types therefore carry the set of locations they may point to, which merging branches can widen (Section 3.4). In (c), the body of `f` mixes its parameter `y` with the captured variable `x` and the captured cell `a`. The two applications `f 1` are syntactically identical, yet the first is typed clean and the second poisoned, because the write between them poisons the cell that `!a` reads. Thus, the type of an application depends on the memory state at the call site and cannot be fixed at the definition, except conservatively as poisoned, losing precision. Function types therefore carry *typing closures*—the body together with its captured typing environment—and bodies are re-typechecked at each application site (Section 3.5).

3 Types for Fault Propagation

Figure 3 shows the syntax of the *Poison* language. It omits extern declarations for brevity; they introduce host-provided external values and functions, and their

$$
\begin{array}{rll}
e ::= & v \mid x \mid e_1\ op\ e_2 \mid \texttt{not}\ e & \text{(values, variables, operations)} \\
\mid & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \text{(conditionals)} \\
\mid & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid e_1; e_2 & \text{(let binding, sequencing)} \\
\mid & \texttt{ref}\ e \mid e_1 := e_2 \mid\ !e & \text{(references)} \\
\mid & (\texttt{fn} \mid \texttt{pfn})\ x{:}\tau \Rightarrow e{:}\tau' \mid e_1\ e_2 & \text{(functions)} \\
v ::= & n \mid \texttt{true} \mid \texttt{false} \mid s \mid \texttt{unit} \mid \langle x, e, \rho \rangle & \\
op ::= & + \mid - \mid \times \mid / \mid \geq \mid \leq \mid \neq \mid > \mid < \mid = \mid \,\&\, \mid \,\&\&\, \mid \,||\, & \\
\tau ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \texttt{string} \mid \texttt{ref}\ \tau \mid \tau_1 \to \tau_2 & \\
& n \in \mathbb{Z} \quad x \in \mathit{Identifiers} \quad s \in \mathit{Strings} \quad \rho \in \mathit{Environments} &
\end{array}
$$

Fig. 3. Syntax of Poison (runtime fragment, without fault-tracking annotations).

concrete syntax is illustrated in Listing 2. Functions written with `fn` are ordinary single-argument functions whose body is an arbitrary expression; those written with `pfn` are *pure* functions, a syntactic subset of `fn` used as de-poisoning predicates and syntactically matched against the conditions of `if` statements. The body of a pure function is limited to side-effect-free expression forms, excluding reference creation, assignment, sequencing, and function definitions; it may, however, dereference existing references, perform arithmetic, use conditionals, and apply previously defined pure functions, which it may also pass and return. Predicates are nevertheless first-order: a guard predicate is unary in the checked value, and guards do not attach to function values. The type system distinguishes pure from impure functions and enforces these restrictions (Section 3.2). A matched check must be pure, so that its outcome depends only on the value it checks and the variables and locations it reads, which is what lets a regime fix that outcome; semantic predicate matching is future work (Section 8). We define a big-step operational semantics for the language, formalised in Lean; the development includes its complete rules. The pure/impure distinction is enforced in the semantics via a *mode* parameter $\pi \in \{\mathit{pure}, \mathit{impure}\}$ threaded through every evaluation rule. Judgements have the form $\langle e, (\pi, \mu, \rho)\rangle \Downarrow \langle v, (\pi, \mu', \rho')\rangle$, meaning that, in mode π, with memory μ and variable environment ρ, expression e evaluates to value v with updated memory μ' and environment ρ'; we write S for a state (π, μ, ρ). A memory μ maps *addresses* to values; references evaluate to addresses, created at run time by `ref`. The type system never sees addresses: it reasons instead about abstract *locations* standing for the cells a reference may denote (Sections 3.1 and 3.2), which the formalisation links to per-run addresses via realisation maps (Section 5).

The type system for the Poison language has several interacting parts (guard tags, regimes, flow-sensitive environments, and typing closures), each motivated by a fault-handling requirement discussed below and revisited in Section 8.

3.1 Surface, Enriched, and Qualified Types

Our type system is inspired by the Smith–Volpano secure information flow type system [44], but uses poison status instead of a security level. It is organised into

three layers: *surface*, *enriched*, and *qualified*. *Surface types* τ are the standard types of Figure 3.

$$
\begin{array}{lll}
T ::= \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \mathsf{string} \mid \mathsf{ref} \mathbin{@} LS \mid \tau_1 \to \tau_2 \triangleright (p,\ tcls) & & \text{(enriched types)} \\
P ::= k,\ y \Rightarrow pe \ \mid \ k & & \text{(guards)} \\
Q ::= \ T \triangleright p\,\{P\} & & \text{(qualified types)}
\end{array}
$$

Fig. 4. Enriched and qualified types.

Enriched types (Figure 4) extend surface types in three ways. First, reference types are annotated with a *points-to set* `@LS`, which is the set of abstract locations a reference may point to, tracked during typechecking as in flow-insensitive points-to analysis [2]. They are needed because a reference may denote several cells, as in Figure 2(b): a write through such a reference must conservatively poison every cell in the set, and a read may rely only on guard facts the whole set shares (Section 3.4). Second, functions are annotated with the *internal arrow poison*, discussed below. Finally, functions are also annotated with *typing closures tcls* (see Section 3.5), which capture function bodies and typing contexts for deferred typechecking at application time: as Figure 2(c) showed, the result of an application depends on the flow-sensitive environments at the call site, so a function's result type cannot be fixed at its definition.

Qualified types $T \triangleright p\,\{P\}$ annotate an enriched type T with a poison status p and an optional guard P for conditional de-poisoning. The underlying enriched type T never changes during typechecking; the poison status p, which may be clean or poisoned, is refined for a given variable or memory location, for example by de-poisoning within a branch. The guard P itself does not change, but may be stripped when de-poisoning is no longer applicable.

A guard P takes two forms, both carrying an identifier called a *guard tag* k. The first form, written $k,\ y \Rightarrow pe$, additionally carries a *guard predicate*, kept as a *predicate closure*: the pure expression pe evaluates to a boolean after capture-avoiding substitution of its formal parameter y with the program expression serving as the predicate's argument, and the type system stores with it the typing environment σ for the other free variables of pe, captured from the definition site. The second, tag-only form, just k, carries no guard predicate. The two forms divide the work: the checked value carries the predicate that a conditional must match, while values validated indirectly by the same check carry only the linking tag. We write an empty annotation $\{\}$ when a type carries no guard. For a guard $P = k,\ y \Rightarrow pe$ and an expression a, we write $P(a)$ for $pe[y := a]$; for $P = 100,\ x \Rightarrow x \geq 0$, $P(\texttt{ret})$ is $\texttt{ret} \geq 0$. Listing 2 used both forms: `ret` has type $\mathsf{int} \triangleright \mathsf{poisoned}\,\{100,\ x \Rightarrow x \geq 0\}$, while the memory location written through `status` carries the tag-only guard $\{100\}$. Initially, both are poisoned and both carry the same guard tag, linking them as a pair. When a subsequent `if` checks the predicate (line 15), the type system matches it against all values sharing tag 100 and de-poisons both `ret` and the cell within the true branch; in the false branch, both remain poisoned.

Table 1. Environments used in the type checking judgement.

$\pi : \{pure, impure\}$	Indicates purity of current context	(mode)
$pc : \mathbb{B}$	Control flow poison status	(pc poison)
$\sigma : Identifiers \rightharpoonup Q$	Partial map from variables to qualified types	(var typ env)
$\Gamma : l \rightharpoonup (p, P)$	Partial map from abstract locations to poison status and guard; the stored type is always `int`	(mem typ env)
$\mathcal{L} : tag \rightharpoonup \mathbb{B} \times LS$	Partial map recording guard-check outcomes and their dependency sets, where $LS \subseteq \mathrm{dom}(\Gamma)$	(logical env)
$\mathcal{R} : tag \rightharpoonup \mathbb{B}$	Regime: partial map from tags to fault status (true = valid, false = faulty)	(regime)

Guard tags are part of the user-provided specification, as indices local to an extern declaration, allocated afresh at each application of an extern function (Section 2): a tag names one validity fact. Giving unrelated values the same tag is a specification error, not a soundness flaw in the type system.

Function types in our system have the shape $\big(\tau_1 \to \tau_2 \triangleright (p', tcls)\big) \triangleright p\,\{P\}$, with two distinct poison flags. The outer flag p is the value-level poison any first-class value carries and records how the value reached this use: a function selected by a poisoned conditional is untrustworthy as data even if both alternatives were created in clean contexts. The inner flag p', the internal arrow poison, is a permanent property of the function that records the program-counter poison at its definition, an implicit flow from the control context into its creation; the fault-robustness proof requires two runs to agree on a function's body and captured environment only when this flag is clean.

In source programs, programmers write surface types in ordinary code and qualified types only on extern declarations; the typechecker infers everything else, including the enriched layer. This is because new poison can only be introduced via external functions and values (Section 2).

3.2 Typing Judgement and Environments

Our type system is flow-sensitive: the base type of a variable or memory location never changes, but its poison status and associated guards can evolve during the analysis. This is what Strom and Yemini [40] call *typestate*, not to be confused with the typestate of object protocols. Our key innovation, de-poisoning, makes poison status changes non-monotonic. Without it, once poisoned a variable could never be made clean, which would make the analysis still sound but less precise.

Our type system proves judgements of the form $\pi; pc; \sigma; \Gamma; \mathcal{L}; \mathcal{R} \vdash e : Q \dashv \Gamma'; \mathcal{L}'; \mathcal{R}'$, where Table 1 defines the components of the judgement. Conceptually, both Γ and σ assign qualified types to their entries; σ stores the full qualified type, whereas Γ tracks only each location's poison status and guard, since only integers can be stored in the heap. The flow-sensitive *environments* Γ, $\mathcal{L}$, and $\mathcal{R}$ are threaded through the typing rules so they may vary at different program points. The remaining components π, pc, and σ stay fixed within a given typing

derivation. In this judgement, the mode π tracks whether the expression is in a pure or impure context, enforcing the side-effect restrictions on pure function bodies described earlier.

Recall that each guard tag links a set of values whose poison status must agree. A *regime* $\mathcal{R}$ is a partial map $\mathcal{R} : \mathit{tag} \rightharpoonup \mathbb{B}$ that specifies, for each tag in its domain, whether the associated values are valid (true) or faulty (false) in a given run; each regime thus encodes one hypothesis about which of the guarded inputs are faulty. In the regimes we consider, its domain is the guard tags of the program's inputs. Our type system is *regime-aware*, carrying $\mathcal{R}$ in the judgement. The conditional rule consumes $\mathcal{R}(k)$: when the regime determines whether tag k is valid or faulty, the values sharing that tag can be de-poisoned in the live branch. Without it, both branches would have to be typed conservatively, poisoning everything that depends on a guarded condition, so a program whose observable output is computed from a guarded value would fail to certify even when it is in fact robust. Like the other flow-sensitive components, $\mathcal{R}$ is threaded through the rules and updated conservatively: a write to a location a guard predicate depends on invalidates the corresponding entry (Section 3.4), and when branches merge, the joined $\mathcal{R}$ retains only the entries on which both branches agree.

The program-counter poison *pc* tracks whether the current control-flow context depends on a poisoned value, analogous to the *pc* label in secure information-flow type systems [44]: when a conditional branches on a poisoned condition, every expression in both branches is tainted by *pc*.

The logical environment $\mathcal{L}$ records which guard checks the program has performed, and with what outcome, at the current point: each entry pairs a guard tag with the boolean outcome of a predicate matched against a condition and with the memory locations that predicate depends on, so writing any of those locations invalidates the entry. Both $\mathcal{L}$ and $\mathcal{R}$ are flow-sensitive maps over guard tags, but they record different things: $\mathcal{R}$ is an assumption about which guarded inputs are faulty, supplied from outside the program, whereas $\mathcal{L}$ is derived from the program's own checks. The two cooperate at a conditional: $\mathcal{R}(k)$ selects the live branch, while the check recorded in $\mathcal{L}$ is what licenses de-poisoning of values sharing tag k at their later uses (Section 3.3).

In the following subsections, we present the most important type rules from our type system; the development includes the complete system.

3.3 Conditional Typing and De-Poisoning

We begin by considering how de-poisoning is handled in the type system by examining the typing rules for conditional expressions.

The conditional rules rely on $\texttt{match_guard}(e_1, \Gamma, \sigma)$, which finds the guard whose predicate, applied to the value it guards, is syntactically the condition e_1. It scans the variables $x \in \mathrm{dom}(\sigma)$; the candidates are the pairs (a, P) below, where the guard must be of the form k, $y \Rightarrow pe$ with pe mentioning y:

$(x,\ P)$ if $\sigma(x) = T \triangleright p\,\{P\}$,
$(!x,\ P_m)$ if $\sigma(x) = \mathsf{ref} \mathbin{@} LS_x \triangleright \mathsf{clean}\,\{_\}$ and $\Gamma(l) = (p_l, P_m)$ for all $l \in LS_x$.

If exactly one candidate has $P(a) = e_1$, the result is (k, deps), where k is the tag of P and *deps* is the *dependency set* of the check: the locations that *pe* reads through its captured environment (possibly none) and, for a reference candidate, the referenced locations LS_x. If no candidate matches, or several do, `match_guard` is undefined and the conditional is typed by the standard rule. A poisoned reference contributes no dereference candidate, since it cannot reliably decide a branch, nor do locations whose guards disagree; the distinct arguments x and $!x$ keep candidates apart.

The function `if_guard_case` classifies a conditional, from `match_guard` and $\mathcal{R}$, as standard, known-true, or known-false. When it is standard, because no guard matches or the matched tag is outside the domain of $\mathcal{R}$, no de-poisoning can occur: both branches are typed under the same environments, with the condition's poison joined into *pc*, and the result carries the join of all three poisons. Since either branch may execute, the branches' updates to Γ, $\mathcal{L}$, and $\mathcal{R}$ are joined in the conclusion, the memory typing with the *strong join* $\sqcup^*$, which keeps entries present in only one branch but strips their guards, so that branch-local guards do not leak into the post-context.

Rule 1 applies when the condition matches a guard tag k set to true by the regime $\mathcal{R}$, so the then-branch will be taken. The first premise binds the matched tag and dependency set; the second selects the known-true regime case.

$$\frac{\begin{array}{c}\texttt{match_guard}(e_1, \Gamma, \sigma) = (k, \mathit{deps}) \qquad \texttt{if_guard_case}(e_1, \Gamma, \sigma, \mathcal{R}) = \mathsf{true} \\ \mathit{pure};\, pc;\, \sigma;\, \Gamma;\, \mathcal{L};\, \mathcal{R} \vdash e_1 : \mathsf{bool} \triangleright p_1\, \{P_1\} \dashv \Gamma;\, \mathcal{L};\, \mathcal{R} \\ \pi;\, pc;\, \sigma;\, \Gamma;\, \mathcal{L}[k \mapsto (\mathsf{true}, \mathit{deps})];\, \mathcal{R} \vdash e_2 : T \triangleright p_2\, \{P_2\} \dashv \Gamma_2;\, \mathcal{L}_2;\, \mathcal{R}_2 \\ \pi;\, pc;\, \sigma;\, \Gamma;\, \mathcal{L}[k \mapsto (\mathsf{false}, \mathit{deps})];\, \mathcal{R} \vdash e_3 : T \triangleright p_3\, \{P_3\} \dashv \Gamma_3;\, \mathcal{L}_3;\, \mathcal{R}_3\end{array}}{\pi;\, pc;\, \sigma;\, \Gamma;\, \mathcal{L};\, \mathcal{R} \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T \triangleright pc \sqcup p_2\, \{\} \dashv \Gamma_2;\, \mathcal{L}_2;\, \mathcal{R}_2} \tag{1}$$

The `match_guard` function only matches pure expressions, so an impure condition cannot trigger de-poisoning and falls through to the standard rule. Because the condition is pure, typing it does not modify Γ or $\mathcal{L}$. The branches are typed under different logical environments: $\mathcal{L}[k \mapsto (\mathsf{true}, \mathit{deps})]$ in the then-branch, enabling de-poisoning of all variables and memory locations with tag k, and $\mathcal{L}[k \mapsto (\mathsf{false}, \mathit{deps})]$ in the else-branch, forcing them to be poisoned. Both branches must have the same enriched type T. Because this case fixes k to true, the conclusion takes its poison ($pc \sqcup p_2$) and environment (Γ_2) from the then-branch only. The else-branch is still typed for correctness, but its effects on Γ and $\mathcal{L}$ are discarded, which is sound under the consistency premise of Section 4.1.

The conditional above establishes $\mathcal{L}(k) = \mathsf{true}$ in the then-branch, but de-poisoning itself is realised when a guarded variable is referenced, or a guarded heap location is dereferenced (Section 3.4), inside that branch. Rule 2 shows the variable case: when $\sigma(x)$ carries a guard with tag k and $\mathcal{L}(k) = \mathsf{true}$, the variable's poison qualifier is ignored and it is typed with poison *pc* (i.e., $\mathsf{false} \sqcup pc$, where false denotes clean). The tag-only variant is identical in its conclusion:

$$\frac{\sigma(x) = T \triangleright p\, \{k,\, y \Rightarrow pe\} \qquad T \neq _ \to _ \qquad \mathcal{L}(k) = (\mathsf{true}, \mathit{deps})}{\pi;\, pc;\, \sigma;\, \Gamma;\, \mathcal{L};\, \mathcal{R} \vdash x : T \triangleright pc\, \{\} \dashv \Gamma;\, \mathcal{L};\, \mathcal{R}} \tag{2}$$

The guard is consumed (replaced with {}): the result value can only acquire further poison from *pc* or from combination with other poisoned operands, neither of which the guard addresses.

These rules are demonstrated by the walkthrough of Section 2. In Listing 2, `match_guard` recognises the check `ret` $\geq$ `0` (line 15) as `ret`'s guard predicate with tag 100; in the regime with $\mathcal{R}(100) = \texttt{true}$, Rule 1 types the `then` branch under $\mathcal{L}[100 \mapsto (\texttt{true}, \mathit{deps})]$, where the heap counterpart of Rule 2 (Section 3.4) de-poisons `!status`, the evolution traced in Figure 1. The known-false counterpart commits to the `else` branch instead.

3.4 Memory Reads and Writes

Heap references complicate fault tracking in ways variables do not. A reference may denote several cells (Figure 2(b)), so the typechecker does not know which cell a read or write touches. A read can therefore use only what every cell in the reference's location set has in common: guards are merged across the set, and a guard survives the merge only where the cells agree on it (when they agree on the tag but not on the predicate closure, the tag alone survives). A write, unless the set is a singleton and the cell can simply be overwritten, is joined into every cell the reference may denote. Poison can also attach to the reference itself rather than to the cell: a poisoned reference has untrusted provenance, so a read through it is poisoned whatever the cell's guard says, and a write through it poisons the target cells and strips their guards.

Checks do not stay valid forever. A guard predicate may read memory, so a write to a location it depends on (as `a := 100` does in Figure 2(a)) invalidates the check: the entries of $\mathcal{L}$ and $\mathcal{R}$ that depend on the written locations are revoked, and a value de-poisoned by that check becomes poisoned again. This revocation is why the regime is threaded through the rules as a flow-sensitive environment rather than fixed in advance. A write under poisoned control flow may not execute on every path, so the guard it would establish is not recorded.

Heap de-poisoning uses the same record as Rule 2: $\mathcal{L}$ records a successful check, and a read through a clean reference whose cells share the checked tag de-poisons the read value, leaving only the control-flow poison *pc*, without rewriting Γ. The read itself leaves Γ, $\mathcal{L}$, and $\mathcal{R}$ as it found them; a write is where they change.

3.5 Function Application

Figure 2(c) showed that function application can require call-site-specific typing. Parametric polymorphism would not achieve this: both calls pass the same clean argument, so any function type quantified over the argument's poison instantiates identically at both sites. What does vary between the sites is the flow-sensitive environments (Γ, $\mathcal{L}$, $\mathcal{R}$), and a function type quantified over them would have to summarise the body's entire effect on them. Instead, as described in Section 3.1, function abstraction does not typecheck the body; it captures

the body, parameter, and variable typing environments in a typing closure. Application rules then re-typecheck the body at each call, extending the captured environment with the actual argument's qualified type. This lets the argument's guard and the caller's environments flow into the body's typing, enabling call-site-specific de-poisoning. For example, let `safe` be `fn v : int ⇒ (if v ≥ 0 then v else 0) : int`. Applied to a poisoned integer with guard $\{k, x \Rightarrow x \geq 0\}$, it yields a clean result whichever value the regime assigns to k: the condition matches the argument's guard, so typing commits to one branch, and both branches yield clean values. Applied to one with guard $\{k', x \Rightarrow x \neq 7\}$, the condition matches nothing and the standard rule poisons the result. Because function types may carry multiple typing closures (accumulated when a function escapes several control-flow paths), application must process each closure independently and combine the results.

4 Fault Robustness

Informally, we want to show that if a program typechecks for all *regimes* consistent with the *input specification*, and the inferred qualified types satisfy the constraints of the *observable specification*, then the program satisfies the *fault-robustness property* for every combination of valid and faulty *critical inputs*.

The *critical inputs* are variables and memory locations that may influence the observable behaviour of the program but can be faulty. The remaining inputs are *non-critical* and expected to be non-faulty. The *observable outputs* (selected memory locations and optionally the return value) are the results we expect not to be affected by input faults; the remaining outputs are unconstrained with respect to input faults. An *input specification* *crit* lists the critical variables ($crit$.vars) and memory locations ($crit$.locs) with their qualified types. The *observable specification* *obs* lists the observable memory locations with their enriched types (obs.locs) and optionally the enriched return type (obs.retEtyp?), which makes the return value observable.

A regime $\mathcal{R} : tag \rightharpoonup \mathbb{B}$ (Section 3.2) assigns each tag a boolean: $\mathcal{R}(k) = \mathsf{true}$ means guard k's predicate holds (the inputs tagged k are valid), false that it fails (they are faulty). For fault-robustness checking, we write K_{crit} for its domain, the guard tags of the critical inputs. Fault robustness must hold for every possible regime, i.e., every assignment in $\mathbb{B}^{K_{crit}}$; since inputs sharing a tag must agree, with $K_{crit} = \{k_1, \ldots, k_n\}$ there are 2^n such assignments. For example, critical inputs x_1, x_2, y tagged k_1, k_1, k_2 give $n = 2$ and hence four regimes, with x_1 and x_2 valid or faulty together. Our enforcement theorem (Section 4.2) is proved once, with the regime universally quantified, but its typechecking premise is regime-specific: certifying a program takes one typechecker run per regime. The typechecker never enumerates regimes, and the exponent n is a property of the input specification, not the program: it counts the distinct fault sources being verified, which we expect to be small in practice; Section 8 sketches a single-pass alternative.

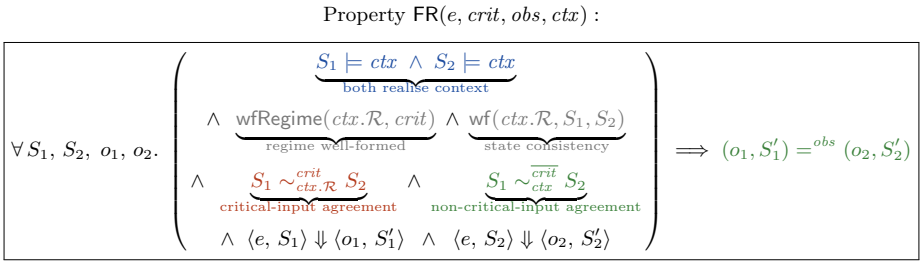


Fig. 5. Fault-robustness property FR (schematic).

4.1 Formalising the Fault Robustness Property

The intuition behind fault robustness is that whether a critical input is faulty may affect the observable outputs, but the value of a faulty input must not leak into them. A value typed *clean* is non-faulty under every regime; a value typed *poisoned* may be valid or faulty depending on the regime.

Formally, fault robustness is regime-relative noninterference [1], a 2-safety property [7]: for a given regime $\mathcal{R}$, any two runs whose input states agree on every input that is *clean under* $\mathcal{R}$ (typed clean, or tagged k with $\mathcal{R}(k) = \mathsf{true}$) must end in final states that agree on observable outputs. Which inputs count as faulty is decided by the regime, not by the program's checks; de-poisoning lets the typechecker treat a checked value as clean. The theorem justifies that trust: a check counts only under the regime in which it holds and only until a write touches a location it depends on (Section 3.4).

Figure 5 gives a schematic presentation of the fault-robustness property FR. Its parameters are the expression e, the input and observable specs *crit* and *obs*, and the typing context *ctx*, which is the tuple of judgement components $(\pi, pc, \sigma, \Gamma, \mathcal{L}, \mathcal{R})$ from Section 3.2. The property considers two runs of e. It quantifies over runtime states S_1, S_2 and their outcomes (returns) o_1, o_2, where a runtime state $S = (\pi, \mu, \rho)$ bundles the mode π with a memory store μ (mapping addresses to values) and a variable environment ρ (cf. Figure 3).

The premises, read top to bottom, are as follows. Both states must *realise* the context: $S \models ctx$ means that the runtime bindings and locations described by *ctx* contain values of the corresponding enriched types. The regime must be well-formed: defined exactly on the critical-input tags ($\mathsf{wfRegime}(ctx.\mathcal{R}, crit)$) and consistent with predicate satisfaction in both states ($\mathsf{wf}(ctx.\mathcal{R}, S_1, S_2)$). The states must be related by *critical-input agreement* $S_1 \sim^{crit}_{ctx.\mathcal{R}} S_2$ and by *non-critical-input agreement* $S_1 \sim^{\overline{crit}}_{ctx} S_2$, which requires clean non-critical entries to agree; both relations are detailed below. Finally, both runs must terminate without a runtime error: the evaluation judgement $\langle e, S\rangle \Downarrow \langle o, S'\rangle$ is the big-step semantics of Section 3, with o the returned value. Under these premises, the runs must produce equal observable outputs: $(o_1, S_1') =^{obs} (o_2, S_2')$ compares the return values and the observable locations designated by *obs*.

The two agreement relations are parameterised differently. Critical-input agreement is parameterised by the input specification *crit*, whose qualified types

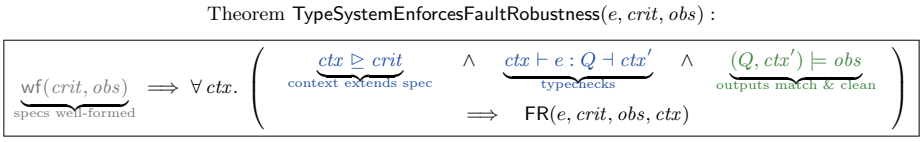


Fig. 6. Typechecking implies FR (schematic).

record which variables and locations carry guard tags, and by the regime $\mathcal{R}$, carried inside the context as its field $ctx.\mathcal{R}$. Non-critical-input agreement ranges over the remaining context entries, $\overline{crit} \triangleq \mathrm{dom}(ctx) \setminus crit$, which is why it is parameterised by the full context ctx rather than only the regime. Both relations state one condition, that the two runs agree on every entry that is clean under $\mathcal{R}$, over two disjoint domains. A critical input tagged k is clean under $\mathcal{R}$ exactly when $\mathcal{R}(k) = \mathsf{true}$, and must then be equal in both runs; an untagged critical input is clean and must always be equal. A non-critical entry carries no critical tag, so for it the condition reduces to: clean entries equal, poisoned entries unconstrained. The predicate obligations, that both runs satisfy the tag-k predicate when $\mathcal{R}(k) = \mathsf{true}$ and both fail it when $\mathcal{R}(k) = \mathsf{false}$, are not part of agreement; they are the consistency premise $\mathsf{wf}(ctx.\mathcal{R}, S_1, S_2)$.

4.2 Type Checking Enforces Fault Robustness

Figure 6 shows our main result schematically, and we discuss its Lean 4 formalisation in Section 5. The theorem states that, provided the input and observable specifications are well-formed ($\mathsf{wf}(crit, obs)$), for any typing context ctx that extends the input specification $crit$ ($ctx \trianglerighteq crit$), if e typechecks under ctx producing qualified return type Q and updated context ctx', and the outputs designated by obs are typed clean ($(Q, ctx') \models obs$: if the return value is observable, Q is clean and of the expected enriched type, and every observable location has a clean entry of its expected type in ctx'), then e satisfies $\mathsf{FR}(e, crit, obs, ctx)$.

The context ctx, and with it the regime $ctx.\mathcal{R}$, is universally quantified: it must include all critical inputs from $crit$ (with guard tags typed as poisoned, untagged ones as clean) but may contain additional non-critical entries.

To illustrate, we instantiate FR for the part of Listing 2 beginning at the outer conditional (line 15), after the `lp55xx_read` call; Table 2 gives the specifications.

The typing context also includes the clean references `status` and `eng` and clean constants such as `READY`, `ERROR`, and `OK`, all non-critical and thus required to agree across runs. The engine cell may start poisoned; the fragment nevertheless meets the observable specification, because every path overwrites `eng` with a clean value. Here $K_{crit} = \{100\}$, so certification takes two typechecker runs, one per assignment of $ctx.\mathcal{R}$, each establishing $(Q, ctx') \models obs$. When $\mathcal{R}(100) = \mathsf{true}$, `ret` $\geq$ `0` de-poisons `ret` and `status`; the inner status test is clean, so both inner arms write `eng` from clean constants and return clean `OK`/`ERROR` codes. When $\mathcal{R}(100) = \mathsf{false}$, the outer `else` branch writes clean `NOT_READY` and returns clean `ERROR`. Thus $\mathsf{FR}(e, crit, obs, ctx)$ holds: the returned code and the engine-state

Table 2. Input and observable specs for part of Listing 2 beginning at line 15.

Component	Contents	Role
crit.vars	$\{\texttt{ret} \mapsto \mathsf{int} \triangleright \mathsf{poisoned}\{100,\ x \Rightarrow x \geq 0\}\}$	critical return code checked by the outer conditional
crit.locs	$\{\ell_{\texttt{status}} \mapsto \mathsf{int} \triangleright \mathsf{poisoned}\{100\}\}$	critical status cell written by `lp55xx_read`
obs.retEtyp?	int	returned code
obs.locs	$\{\ell_{\texttt{eng}} \mapsto \mathsf{int}\}$	engine-state cell

cell may depend on whether `lp55xx_read` failed, but not on the return code and status-cell value it produces when it fails.

5 Formalisation and Proofs in Lean 4

We formalised the operational semantics, type system, fault-robustness property, and our main theorem in Lean 4. The top-level theorem, `TypeSystemEnforces-FaultRobustness` (sketched in Figure 6), is stated about the executable type-checker itself. Further hypotheses require the typing context to be well-formed and execution to start with a clean program-counter and an empty tag environment. Two additional parameters treat predicate satisfaction as an abstract relation. They are universally quantified and constrained by consistency laws so that the theorem holds for every conforming interpretation.

The full statement is additionally parameterised by an *extern registry*, which supplies the host implementations behind extern declarations together with contracts they must satisfy; Section 8 discusses its status.

The formalisation distinguishes the abstract locations the type system reasons about from the physical addresses of each run, connecting them through per-run *realisation maps*; observable outputs are compared up to extensions of these maps that account for fresh allocations.

The complete proof of this theorem is approximately 94 KLOC of Lean 4 (the whole development about 98 KLOC), counting non-blank, non-comment lines. Most of it is a long but routine induction on expression structure; the genuine complications came from a few sources: two-run agreement must be conditioned on the regime, typing-time predicates must be bridged to runtime satisfaction, the induction requires mutually dependent invariants that reach inside captured closures, and the two runs may diverge in allocation and control flow.

6 Experimental Results

The development includes a test suite of 111 Poison programs with their expected outputs. They cover arithmetic and boolean expressions, conditionals,

references that may denote several cells, first-class functions (pure and impure, some carrying several typing closures), and extern values, functions and references with guards and effects. Most target fault tracking: guard tags shared by values and heap cells, de-poisoning by matched checks, invalidation of a check by a later write, regimes, and poisoned arguments to externs that require clean ones. The suite includes the program of Listing 2 under both outcomes of its check, and the buggy variant (Listing 1), whose outputs are inferred poisoned.

The test harness runs the typechecker and then the interpreter on each program and compares the output with the expected one: a type error, or the inferred qualified type and memory typing, which record which values are clean, followed by the evaluation result. Of these programs, 28 are expected to be rejected (two of them by the parser, for ill-formed extern declarations), and 46, once type checking succeeds, stop at run time at an extern that has no implementation in the interpreter. All 111 produce their expected output.

7 Related Work

Information-flow and noninterference. Our work builds on language-based information-flow security. Classical information flow type systems assign security labels to values and show that typechecking enforces a relational noninterference property [44,36]. We reuse this structure but reinterpret the lattice as *clean* versus *poisoned* to model fault propagation rather than secrecy. As in prior relational formulations [1], our guarantee is stated over two runs, but it requires agreement only on designated observable outputs, under regimes that determine which critical inputs are faulty. Our system is flow-sensitive in the sense of Hunt and Sands [23]. Most information-flow systems provide *declassification*, marking a value as having a lower security level than its computed type; these steps are often ad hoc, requiring separate reasoning to ensure they are placed correctly [37]. In contrast, fault status in Poison is non-monotone: a value becomes clean again after the program's own check matches its declared guard, and the same check recovers correlated return values and heap locations through a shared tag.

Abstract interpretation. Abstract non-interference parametrises noninterference by abstract domains for what an attacker observes and what may be released [21]; in its terms, fault robustness releases whether each critical input is faulty, but no faulty payload. Urban and Müller derive sound analyses of input data usage, that is, of inputs on which a program's outcome, non-termination included, does not depend, and argue that noninterference analyses such as the flow-sensitive type system of Hunt and Sands [23] are abstractions of their dependency semantics for terminating programs [43]. Fault robustness is a termination-insensitive, regime-relative form of this independence, and our typing adds recovery after the program's own checks. Termination resilience, also proved by abstract interpretation [33], rules out non-termination forced by untrusted inputs, which fault robustness leaves aside. Kovács et al. verify 2-hypersafety properties, a class that includes fault robustness, by abstract interpretation of self-composed control-flow graphs [29]; this can be more precise than a program-counter label

that taints both branches of a secret-dependent conditional, a taint Poison avoids when the regime fixes a matched guard's outcome. Our judgement is a flow- and context-sensitive analysis with a machine-checked soundness proof; Poison has higher-order functions and aliased references, which the languages formalised in [43,33,29] lack, but not yet their loops.

Taint and dependency analysis. Taint and dependency analyses are structurally close to our work: marks propagate along data flow as poison does, and marked dependencies can be sanitised, by an explicit semantic operation [27] or by checks that a run-time monitor trusts to clear the taint [39]. In Poison, de-poisoning happens only where the program's own check matches the declared guard predicate, and the predicate's meaning is carried into the fault-robustness property (Section 4). Keil and Thiemann's guarantee is termination-insensitive noninterference for marked sources; ours is fault robustness, independence from faulty payloads at observable outputs. Pure dynamic taint analysis also does not compute control-dependent flows on its own, whereas Poison does.

Flow-sensitive refinement. Our type structure also relates to flow-sensitive qualifiers, occurrence typing, and typestate. As in type qualifier systems, our analysis decorates ordinary types with an additional qualifier updated as control flows through the program [17,18]. As in occurrence typing, conditionals refine the information available in each branch [42,28,6]. Our refinement is narrower, because de-poisoning is triggered by syntactic matching against explicit guard predicates rather than a general logical theory, a limitation we return to in Section 8; it is richer because one check can de-poison several correlated values, including heap-resident state, and because that information is propagated through higher-order functions via typing closures. Like typestate [40,16,4,19], we enrich types with state that changes across program points, but our state is the poison status of values and memory locations rather than an object protocol, it changes non-monotonically when a check succeeds, and the typechecker infers it, certifying fault isolation rather than checking that each operation is legal.

Refinement types. The most directly comparable refinement-type system is RefinedC [38], which targets a similar domain, is also mechanised, and automates type checking; its Figure 1 shows exactly the sentinel-plus-result pattern, expressed through abstract types grounded in separation logic. RefinedC asks for per-function specifications and data-structure and loop invariants, and verifies each program by a foundational Rocq proof, some of whose verification conditions must be discharged by hand. Poison asks only for a few input/output annotations, because its property is fixed, and we prove once in Lean that typechecking implies fault robustness, so any program is checked by running the typechecker (once per regime), with no per-program proof. In exchange, RefinedC covers a far richer language and proves much stronger functional properties, including the controlled dependence on the error code, a different result per failure, that Poison deliberately excludes (Section 2).

Error handling analyses. Prior work analyses how errors propagate through systems code and finds mistakes in error-handling paths [22,35,24,47]. These are primarily bug-finding analyses. Our work moves from detection to certifi-

cation: successful typechecking under an input/output specification guarantees that faults in designated inputs cannot affect designated outputs.

8 Conclusions

We have presented a type-system-based approach to fault-propagation and isolation analysis in which an executable typechecker certifies fault robustness: we have formulated a formal two-run relational fault-robustness property, parameterised by regimes, and proved in Lean 4 that typechecking conservatively enforces it, providing compile-time guarantees without runtime verification. The formal semantics, definitional interpreter, and typechecker are all mechanised in Lean 4, and the main theorem is stated about the executable typechecker itself, so the type checking tool can serve as a certified fault-isolation verifier. The full theorem is additionally parameterised by an extern registry (Section 5): we have implemented and proved a RustBelt [26]-style fault-robustness contract layer for extern functions, but leave its presentation to future work.

To certify programs that check a guard and then recover correlated values, the type system needs guard tags to connect a checked value with the payloads and heap locations it validates; regimes to type one possible valid/faulty world precisely while the fault-robustness theorem ranges over all such worlds; flow-sensitive environments to carry de-poisoning across program points and revoke it after writes a checked guard depends on; location sets so that dereferences through aliases observe intervening writes; and typing closures to re-typecheck a function's possible bodies at each application. Typing closures are a precision/-compositionality tradeoff: surface interfaces still compose, but the current core does not give a fully compositional fault-robustness summary for every function value.

The language is intentionally small, but it covers mutable heap references and higher-order functions with closures, features that make fault-isolation analysis non-trivial and are pervasive in real systems code. Several extensions would move Poison from a mechanised core calculus toward a certified fault-isolation verifier for languages such as C. The current guard matcher is syntactic; extending it to semantic matching, so that a check logically equivalent to, or implying, the declared guard predicate also de-poisons, could build on canonical Boolean representations [5], equality saturation [41,46], SMT solving [32], and predicate abstraction [3,31,30]. Multiple guard tags per value would model data whose validity depends on several sources, and distinguishing several classes of faulty value would cover APIs where different failures must be reported differently. Loops and recursion require a fixpoint treatment of the flow-sensitive environments, while closer C-like targets require pointer arithmetic, structs, arrays, and richer alias summaries. Finally, certification currently takes one typechecker run per regime (Section 4); a single pass could instead thread a family of flow-sensitive environments indexed by partial regimes, split only where the regime is consulted.

The accompanying development includes the Lean 4 formalisation of the Poison language, operational semantics, type system, executable typechecker, and fault-robustness property, `sorry`-free proofs of the main theorem, and the complete specification of the semantics and typing rules. The formalisation does not use any non-standard axioms. The development is available in a public GitHub repository at https://github.com/vzaliva/poison-lang.

References

1. Amtoft, T., Banerjee, A.: Information flow analysis in logical form. In: Giacobazzi, R. (ed.) Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3148, pp. 100–115. Springer, Verona, Italy (2004). https://doi.org/10.1007/978-3-540-27864-1_10
2. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Phd thesis, University of Copenhagen (DIKU), Copenhagen, Denmark (1994)
3. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Burke, M., Soffa, M.L. (eds.) Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001. pp. 203–213. ACM (2001). https://doi.org/10.1145/378795.378846
4. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S. (eds.) Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada. pp. 301–320. ACM (2007). https://doi.org/10.1145/1297027.1297050
5. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992). https://doi.org/10.1145/136035.136043
6. Castagna, G., Lanvin, V., Laurent, M., Nguyen, K.: Revisiting occurrence typing. Sci. Comput. Program. **217**, 102781 (2022). https://doi.org/10.1016/J.SCICO.2022.102781
7. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010). https://doi.org/10.3233/JCS-2009-0393
8. CVE-2002-1372. National Vulnerability Database (2002), https://nvd.nist.gov/vuln/detail/CVE-2002-1372, CUPS does not check the return values of file and socket operations, leaking file descriptors
9. CVE-2017-6964. National Vulnerability Database (2017), https://nvd.nist.gov/vuln/detail/CVE-2017-6964, `dmcrypt-get-device` (Debian and Ubuntu) does not check the return values of `setuid` and `setgid`
10. CVE-2019-15900. National Vulnerability Database (2019), https://nvd.nist.gov/vuln/detail/CVE-2019-15900, `doas` does not check the result of `sscanf` and runs the command as root
11. CVE-2019-5818. National Vulnerability Database (2019), https://nvd.nist.gov/vuln/detail/CVE-2019-5818, Google Chrome overlooks the status returned by `ReadBits`, exposing uninitialised process memory
12. CVE-2020-17533. National Vulnerability Database (2020), https://nvd.nist.gov/vuln/detail/CVE-2020-17533, Apache Accumulo does not check the return values of policy enforcement functions before permitting administrative operations

13. CVE-2024-58075. National Vulnerability Database (2025), https://nvd.nist.gov/vuln/detail/CVE-2024-58075, the Linux Tegra crypto driver transfers a request after its initialisation fails
14. CVE-2025-25724. National Vulnerability Database (2025), https://nvd.nist.gov/vuln/detail/CVE-2025-25724, `libarchive` does not check the return value of `strftime`
15. DeFreez, D., Baldwin, H.M., Rubio-González, C., Thakur, A.V.: Effective error-specification inference via domain-knowledge expansion. In: Dumas, M., Pfahl, D., Apel, S., Russo, A. (eds.) Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019. pp. 466–476. ACM (2019). https://doi.org/10.1145/3338906.3338960
16. DeLine, R., Fähndrich, M.: Typestates for objects. In: Odersky, M. (ed.) European Conference on Object-Oriented Programming, ECOOP 2004, Oslo, Norway, June 14-18, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3086, pp. 465–490. Springer (2004). https://doi.org/10.1007/978-3-540-24851-4_21
17. Foster, J.S., Fähndrich, M., Aiken, A.: A theory of type qualifiers. In: Ryder, B.G., Zorn, B.G. (eds.) Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999. pp. 192–203. ACM (1999). https://doi.org/10.1145/301618.301665
18. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI 2002, Berlin, Germany, June 17-19, 2002. pp. 1–12. ACM (2002). https://doi.org/10.1145/512529.512531
19. Garcia, R., Tanter, É., Wolff, R., Aldrich, J.: Foundations of typestate-oriented programming. ACM Trans. Program. Lang. Syst. **36**(4), 12:1–12:44 (2014). https://doi.org/10.1145/2629609
20. Garreton, G., Barr, V.: Phd forum of the 2011 grace hopper celebration of women in computing. In: Garreton, G., Barr, V. (eds.) PhD Forum of the Grace Hopper Celebration of Women in Computing. pp. 1–150. Anita Borg Institute for Women and Technology, Anita Borg Institute, Portland, Oregon (November 2011)
21. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: parameterizing non-interference by abstract interpretation. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 186–197. POPL '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/964001.964017
22. Gunawi, H.S., Rubio-González, C., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Liblit, B.: EIO: error handling is occasionally correct. In: Baker, M., Riedel, E. (eds.) 6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA. pp. 207–222. USENIX, San Jose, CA, USA (2008), http://www.usenix.org/events/fast08/tech/gunawi.html
23. Hunt, S., Sands, D.: On flow-sensitive security types. In: Morrisett, J.G., Jones, S.L.P. (eds.) Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006. pp. 79–90. ACM, Charleston, South Carolina, USA (2006). https://doi.org/10.1145/1111037.1111045
24. Jana, S., Kang, Y.J., Roth, S., Ray, B.: Automatically detecting error handling bugs using error specifications. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 345–362. USENIX Association (2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/jana

25. Jia, Z., Li, S., Yu, T., Liao, X., Wang, J., Liu, X., Liu, Y.: Detecting error-handling bugs without error specification input. In: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019. pp. 213–225. IEEE (2019). https://doi.org/10.1109/ASE.2019.00029
26. Jung, R., Jourdan, J., Krebbers, R., Dreyer, D.: Rustbelt: securing the foundations of the rust programming language. Proc. ACM Program. Lang. **2**(POPL), 66:1–66:34 (2018). https://doi.org/10.1145/3158154
27. Keil, M., Thiemann, P.: Type-based dependency analysis for javascript. In: Naldurg, P., Swamy, N. (eds.) Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013, Seattle, WA, USA, June 20, 2013. pp. 47–58. ACM (2013). https://doi.org/10.1145/2465106.2465118
28. Kent, A.M., Kempe, D., Tobin-Hochstadt, S.: Occurrence typing modulo theories. In: Krintz, C., Berger, E.D. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 296–309. ACM (2016). https://doi.org/10.1145/2908080.2908091
29. Kovács, M., Seidl, H., Finkbeiner, B.: Relational abstract interpretation for the verification of 2-hypersafety properties. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. pp. 211–222. CCS '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2508859.2516721
30. Lahiri, S.K., Ball, T., Cook, B.: Predicate abstraction via symbolic decision procedures. In: Etessami, K., Rajamani, S.K. (eds.) Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings. pp. 24–38. Lecture Notes in Computer Science, Springer (2005). https://doi.org/10.1007/11513988_5
31. Lahiri, S.K., Nieuwenhuis, R., Oliveras, A.: SMT techniques for fast predicate abstraction. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings. pp. 424–437. Lecture Notes in Computer Science, Springer (2006). https://doi.org/10.1007/11817963_39
32. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. pp. 337–340. Lecture Notes in Computer Science, Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
33. Moussaoui Remil, N., Urban, C.: Termination Resilience Static Analysis. In: Verification, Model Checking, and Abstract Interpretation, VMCAI 2026. pp. 212–236. Lecture Notes in Computer Science, Springer (2026). https://doi.org/10.1007/978-3-032-15700-3_11
34. Potter, P.: leds: lp5523: check return value of lp5xx_read and jump to cleanup code. Linux kernel patch (May 2021), https://lkml.iu.edu/2105.0/00782.html, fix for commit 248b57015f35; `drivers/leds/leds-lp5523.c`
35. Rubio-González, C., Gunawi, H.S., Liblit, B., Arpaci-Dusseau, R.H., Arpaci-Dusseau, A.C.: Error propagation analysis for file systems. In: Hind, M., Wendtland, A.P. (eds.) Proceedings of the 30th ACM SIGPLAN Conference on Program-

ming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009. pp. 270–280. ACM (2009). https://doi.org/10.1145/1542476.1542506
36. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003). https://doi.org/10.1109/JSAC.2002.806121
37. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. J. Comput. Secur. **17**(5), 517–548 (2009). https://doi.org/10.3233/JCS-2009-0352
38. Sammler, M., Lepigre, R., Krebbers, R., Memarian, K., Dreyer, D., Garg, D.: Refinedc: automating the foundational verification of C code with refined ownership types. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 158–174. ACM (2021). https://doi.org/10.1145/3453483.3454036
39. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 31st IEEE Symposium on Security and Privacy, SP 2010, 16-19 May 2010, Berleley/Oakland, California, USA. pp. 317–331. IEEE Computer Society (2010). https://doi.org/10.1109/SP.2010.26
40. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Software Eng. **12**(1), 157–171 (1986). https://doi.org/10.1109/TSE.1986.6312929
41. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: Shao, Z., Pierce, B.C. (eds.) Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009. pp. 264–276. ACM (2009). https://doi.org/10.1145/1480881.1480915
42. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 395–406. ACM (2008). https://doi.org/10.1145/1328897.1328486
43. Urban, C., Müller, P.: An Abstract Interpretation Framework for Input Data Usage. In: Programming Languages and Systems, ESOP 2018. pp. 683–710. Lecture Notes in Computer Science, Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_24
44. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2/3), 167–188 (1996). https://doi.org/10.3233/JCS-1996-42-304
45. Weiss, C., Rubio-González, C., Liblit, B.: Database-backed program analysis for scalable error propagation. In: Bertolino, A., Canfora, G., Elbaum, S.G. (eds.) 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1. pp. 586–597. IEEE Computer Society (2015). https://doi.org/10.1109/ICSE.2015.75
46. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: egg: Fast and extensible equality saturation. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434304
47. Wu, Q., Pakki, A., Emamdoost, N., McCamant, S., Lu, K.: Understanding and detecting disordered error handling with precise function pairing. In: Bailey, M.D., Greenstadt, R. (eds.) 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021. pp. 2041–2058. USENIX Association, Online (2021), https://www.usenix.org/conference/usenixsecurity21/presentation/wu-qiushi

Appendix A Extended Presentation of the Semantics, Type System, and Proofs

This appendix is intended to be read after the main paper, to gain additional deeper insight into the design of the type system and the details of the Lean mechanisation. The table below relates each of its parts to the sections of the paper it complements. The complete definition of the language, with every evaluation and typing rule, is part of the development.

Appendix	Contents	Sections
A.1	Two sample evaluation rules, B-If-True and B-Let (Figure 7)	3
A.2	The typing rules that the body only describes: the standard conditional (Rule 3), memory reads and writes (Rules 4 and 5), and function application (Rules 6 and 7)	3.3–3.5
A.3	The main theorem with its hypotheses (Listing 3) and the fault-robustness property (Listing 4) in Lean, a table mapping the schematic notation to Lean names (Table 3), realisation maps, and the Lean representation of guards	3.1, 4.1, 4.2, 5
A.4	The proof techniques behind the complications named in Section 5:	5
A.4.1	two-run agreement conditioned on the regime (Listing 5), including the internal arrow poison	3.1, 4.1
A.4.2	typing-time predicates bridged to runtime satisfaction	
A.4.3–A.4.4	mutually dependent invariants that reach inside captured closures	
A.4.5–A.4.6	runs that diverge in allocation and control flow	

A.1 Sample Evaluation Rules

Figure 7 shows two of the big-step evaluation rules whose judgement form Section 3 introduces. In B-If-True, the condition e_1 evaluates to `true`, so the then-branch e_2 is evaluated starting from the memory and environment produced by the condition, and its result is that of the whole conditional; a symmetric rule (not shown) handles a `false` condition. In B-Let, e_1 is evaluated first, and the body e_2 is then evaluated with x bound to e_1's value; the conclusion restores the original environment ρ, as the binding is local to the `let`.

A.2 Walkthrough of Representative Typing Rules

The standard conditional. The "standard" conditional rule (Rule 3) applies when `if_guard_case` is standard: either `match_guard` does not find a match, or the

$$\frac{\langle e_1, (\pi, \mu, \rho)\rangle \Downarrow \langle \texttt{true}, (\pi, \mu_1, \rho_1)\rangle \qquad \langle e_2, (\pi, \mu_1, \rho_1)\rangle \Downarrow \langle v, (\pi, \mu_2, \rho_2)\rangle}{\langle \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3,\ (\pi, \mu, \rho)\rangle \Downarrow \langle v, (\pi, \mu_2, \rho_2)\rangle}\ \text{B-If-True}$$

$$\frac{\langle e_1, (\pi, \mu, \rho)\rangle \Downarrow \langle v, (\pi, \mu_1, \rho_1)\rangle \qquad \langle e_2, (\pi, \mu_1, \rho[x \mapsto v])\rangle \Downarrow \langle v_2, (\pi, \mu_2, \rho_2)\rangle}{\langle \texttt{let } x = e_1 \texttt{ in } e_2,\ (\pi, \mu, \rho)\rangle \Downarrow \langle v_2, (\pi, \mu_2, \rho)\rangle}\ \text{B-Let}$$

Fig. 7. Two sample big-step evaluation rules.

matched guard's tag is not in the domain of $\mathcal{R}$, so the regime does not determine its outcome. Without a known guard outcome, no de-poisoning can occur, so both branches are typed under the same environments.

$$\frac{\begin{array}{c}\texttt{if_guard_case}(e_1, \Gamma, \sigma, \mathcal{R}) = \texttt{std} \\ \pi; pc; \sigma; \Gamma; \mathcal{L}; \mathcal{R} \vdash e_1 : \mathsf{bool} \triangleright p_1\ \{P_1\} \dashv \Gamma_1; \mathcal{L}_1; \mathcal{R}_1 \\ \pi; pc \sqcup p_1; \sigma; \Gamma_1; \mathcal{L}_1; \mathcal{R}_1 \vdash e_2 : T \triangleright p_2\ \{P_2\} \dashv \Gamma_2; \mathcal{L}_2; \mathcal{R}_2 \\ \pi; pc \sqcup p_1; \sigma; \Gamma_1; \mathcal{L}_1; \mathcal{R}_1 \vdash e_3 : T \triangleright p_3\ \{P_3\} \dashv \Gamma_3; \mathcal{L}_3; \mathcal{R}_3\end{array}}{\pi; pc; \sigma; \Gamma; \mathcal{L}; \mathcal{R} \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : T \triangleright pc \sqcup p_1 \sqcup p_2 \sqcup p_3\ \{\} \dashv \Gamma_2 \sqcup^* \Gamma_3; \mathcal{L}_2 \sqcup \mathcal{L}_3; \mathcal{R}_2 \sqcup \mathcal{R}_3} \quad (3)$$

Since no known guard outcome is available, the type system treats the condition conservatively: the condition's poison flows into both branches via $pc \sqcup p_1$, and all three poisons are joined in the result. Since either branch may execute, their updates to Γ, $\mathcal{L}$, and $\mathcal{R}$ are joined in the conclusion. The memory typing uses the *strong join* $\sqcup^*$, defined location-wise: for entries present in both branches, poisons are joined and guards are merged by tag agreement (as in Rule 4). For entries present in only one branch, the entry is kept but its guard is stripped to the empty annotation $\{\}$, ensuring that branch-local guards do not leak into the post-context.

Rules 1 and 3 are two representative non-reference/non-function rules: the known-true case and the standard case. The full system has eight variants: four result shapes (ordinary, reference, impure function, pure function) times two known regime outcomes. Unknown regimes use variants of the standard rule; reference variants union location sets, and function variants merge closure sets.

Memory reads. Dereference typing is important because it is where a memory cell's de-poisoning takes effect. However, several complications arise: the reference may alias several cells, as in Figure 2(b), and poison can attach to the reference itself, not just to the cell it denotes. The rules split along three axes: reference poison (clean vs. poisoned), the merged guard across the location set (absent vs. present), and, when a guard is present, its status in $\mathcal{L}$ (true, false, or unknown), yielding eight rules in total. Dereferencing a poisoned reference unconditionally yields a poisoned result with no guard: the reference's untrusted provenance dominates any guard stored in the target cell. Unlike assignment, dereference does not modify Γ, $\mathcal{L}$, or $\mathcal{R}$: it is a read of the memory typing.

We present the guard-true case for clean references, where de-poisoning occurs. This rule applies when agreement-only merge of guards across all locations in LS produces a guard tag k that is true in $\mathcal{L}$. The cell poison is then discarded,

but a poisoned pc still overrides the de-poisoning:

$$\frac{\pi; pc; \sigma; \Gamma; \mathcal{L}; \mathcal{R} \vdash e : \mathsf{ref} \mathbin{@} LS \triangleright \mathsf{clean}\,\{P\} \dashv \Gamma_1; \mathcal{L}_1; \mathcal{R}_1 \quad \bigsqcup_{l \in LS} \mathrm{pred}(\Gamma_1(l)) = (k,\, y \Rightarrow pe)\ ① \quad \mathcal{L}_1(k) = (\mathsf{true}, deps)\ ②}{\pi; pc; \sigma; \Gamma; \mathcal{L}; \mathcal{R} \vdash\, !e : \mathsf{int} \triangleright pc\,\{\} \dashv \Gamma_1; \mathcal{L}_1; \mathcal{R}_1} \tag{4}$$

A few notable aspects of the rule:

- **Location-set merging** ①. Guards are merged across all $l \in LS$ before the guard is consulted: any of these locations may be the read cell, so only guard facts they all share can be soundly used. The merge is agreement-only: if all entries share the same guard exactly, it is preserved; if they share the same tag but differ in the closure, only the tag is kept, so the read value is de-poisoned once the tag is checked but cannot itself be matched as that check; if tags disagree, the guard is dropped.
- **Guard-directed de-poisoning** ②. A successful check is recorded once in $\mathcal{L}$, and each read consults that record, so a single check covers every cell sharing the tag without rewriting Γ: dereference is where value-level de-poisoning is realised. Because $\mathcal{L}_1(k) = \mathsf{true}$, the cell poison is discarded entirely and the result carries only pc. The guard is consumed (replaced with $\{\}$), since no further de-poisoning is possible.

Memory writes. These rules matter because they involve several aspects of poison handling at once: a write can undermine an earlier check whose predicate reads the written cell (as `a := 100` does in Figure 2(a)); the reference being written may denote several cells (Figure 2(b)); and a write under poisoned control flow may not execute on every path. Assignment typing splits into four rules along two axes: reference poison (clean vs. poisoned) and whether the reference's points-to set LS is a singleton (the reference denotes exactly one cell) or not. A poisoned reference originates from untrusted data, so any assignment through it unconditionally poisons the corresponding entry in Γ and strips its guard, blocking later de-poisoning. Singleton location sets admit strong updates that overwrite the entry directly; non-singleton sets require a conservative join over all possible targets. We present the clean, non-singleton case, which already exhibits all three key mechanisms: guard invalidation, location-set joining, and predicate gating.

$$\frac{\begin{array}{c}\pi; pc; \sigma; \Gamma; \mathcal{L}; \mathcal{R} \vdash e_1 : \mathsf{ref} \mathbin{@} LS \triangleright \mathsf{clean}\,\{P_1\} \dashv \Gamma_1; \mathcal{L}_1; \mathcal{R}_1 \\ \pi; pc; \sigma; \Gamma_1; \mathcal{L}_1; \mathcal{R}_1 \vdash e_2 : \mathsf{int} \triangleright p_2\,\{P_2\} \dashv \Gamma_2; \mathcal{L}_2; \mathcal{R}_2 \\ \texttt{invalidate_deps}(\mathcal{L}_2, LS) = \mathcal{L}_3 \qquad \texttt{invalidate_regime_deps}(\mathcal{R}_2, \mathcal{L}_2, LS) = \mathcal{R}_3\ ① \\ p_3 = \bigsqcup_{l \in LS} \mathrm{poison}(\Gamma_2(l)) \qquad P_3 = \bigsqcup_{l \in LS} \mathrm{pred}(\Gamma_2(l))\ ② \\ P_4 = (\text{if } pc = \mathsf{clean} \text{ then } P_2 \text{ else } \{\}) \sqcup P_3\ ③\end{array}}{\pi; pc; \sigma; \Gamma; \mathcal{L}; \mathcal{R} \vdash e_1 := e_2 : \mathsf{unit} \triangleright pc\,\{\} \dashv \Gamma_2[LS \mapsto (p_3 \sqcup pc \sqcup p_2)\,\{P_4\}]; \mathcal{L}_3; \mathcal{R}_3} \tag{5}$$

The three mechanisms work as follows:

- **Guard invalidation** ①. Guard predicates may reference memory locations that the assignment modifies, so any prior determination of their satisfaction is no longer valid. The `invalidate_deps` and `invalidate_regime_deps`

premises therefore revoke every entry in $\mathcal{L}$ and $\mathcal{R}$ whose dependency set intersects LS, so no stale guard can justify a subsequent de-poisoning. A value de-poisoned by an earlier check can therefore become poisoned again once a write touches a location its guard depends on. A fixed clean/poisoned labelling cannot express this toggling, which is why the regime is threaded as a flow-sensitive environment rather than fixed in advance (Section 3.2).

– **Location-set joining** ②. As in the dereference typing rules (Rule 4), since the reference may denote any location in LS, no single entry can be updated in isolation. The rule joins both poison and guard state across all locations in LS via the same lattice and agreement-only joins, then merges the new typing into that summary.
– **Predicate gating** ③. An assignment under poisoned control flow may not execute on all paths. Retaining its guard would be unsound, so the rule strips it when pc is poisoned.

Function application. Section 3.5 explains why application re-typechecks the body of every typing closure at the call site. The rules vary by purity (impure vs. pure), by the shape of the result type (base type, reference, or arrow), and by whether the closure set is a singleton (base case) or is peeled off recursively (cons case), yielding twelve total application rules.

We illustrate the function application typing rules with the impure variant for base types. The two rules below define a recursive traversal of the closure list: the base case (Rule 6) handles a singleton closure, and the recursive case (Rule 7) peels off the head closure and applies the tail recursively. Both rules share a common structure: the argument e_2 is typed first (the semantics evaluates it before the function), then the function e_1 is typed to extract the closure set, and the head closure's body is re-typechecked in its captured environment extended with the argument's qualified type ①. The premises $\lfloor T_1 \rfloor = \tau_1$ and $\lfloor T_2 \rfloor = \tau_2$ require the enriched types of the argument and of the body's result to erase to the arrow's surface types, where $\lfloor T \rfloor$ drops the location sets, internal arrow poisons, and closure sets of T.

$$\frac{\begin{array}{c} \mathit{impure}; pc; \sigma; \Gamma; \mathcal{L}; \mathcal{R} \vdash e_2 : T_1 \triangleright p_2 \, \{P_2\} \dashv \Gamma_2; \mathcal{L}_2; \mathcal{R}_2 \\ \mathit{impure}; pc; \sigma; \Gamma_2; \mathcal{L}_2; \mathcal{R}_2 \vdash e_1 : \tau_1 \to \tau_2 \triangleright (p,\ \{(x, e, \sigma_1)\}) \triangleright p_1 \, \{P_1\} \dashv \Gamma_1; \mathcal{L}_1; \mathcal{R}_1 \\ \lfloor T_1 \rfloor = \tau_1 \qquad \lfloor T_2 \rfloor = \tau_2 \qquad p_6 = p_1 \sqcup p \\ \mathit{impure}; p \sqcup p_1 \sqcup pc; \sigma_1[x \mapsto T_1 \triangleright pc \sqcup p_2 \, \{P_2\}]\ ①; \Gamma_1; \mathcal{L}_1; \mathcal{R}_1 \vdash e : T_2 \triangleright p_3 \, \{P_3\} \dashv \Gamma_3; \mathcal{L}_3; \mathcal{R}_3 \\ P_4 = (\text{if } p_6 = \mathsf{clean} \text{ then } P_3 \text{ else } \{\}) \end{array}}{\mathit{impure}; pc; \sigma; \Gamma; \mathcal{L}; \mathcal{R} \vdash e_1\ e_2 : T_2 \triangleright pc \sqcup p_1 \sqcup p \sqcup p_3 \, \{P_4\} \dashv \Gamma_3; \mathcal{L}_3; \mathcal{R}_3} \tag{6}$$

In the singleton case, the body's result is the application's result directly. The recursive case (Equation 7) additionally applies the remaining closures $tcls'$ to the same argument. It does so through a fresh metavariable e_0 ② — a function with the same type as e_1 but carrying only the tail closures $tcls'$. The head and tail results are required to be signature-compatible ③, and their poisons, guards, and post-environments are joined in the conclusion.

$$\frac{\begin{array}{c} \mathit{impure}; pc; \sigma; \Gamma; \mathcal{L}; \mathcal{R} \vdash e_2 : T_1 \triangleright p_2\,\{P_2\} \dashv \Gamma_2; \mathcal{L}_2; \mathcal{R}_2 \\ \mathit{impure}; pc; \sigma; \Gamma_2; \mathcal{L}_2; \mathcal{R}_2 \vdash e_1 : \tau_1 \to \tau_2 \triangleright (p,\, \{(x, e, \sigma_1)\} \cup \mathit{tcls}') \triangleright p_1\,\{P_1\} \dashv \Gamma_1; \mathcal{L}_1; \mathcal{R}_1 \\ \mathit{impure}; pc; \sigma; \Gamma_2; \mathcal{L}_2; \mathcal{R}_2 \vdash e_0 \ \textcircled{2} : \tau_1 \to \tau_2 \triangleright (p,\, \mathit{tcls}') \triangleright p_1\,\{P_1\} \dashv \Gamma_1; \mathcal{L}_1; \mathcal{R}_1 \\ \lfloor T_1 \rfloor = \tau_1 \qquad \mathit{tcls}' \neq \emptyset \\ \mathit{impure}; p \sqcup p_1 \sqcup pc; \sigma_1[x \mapsto T_1 \triangleright pc \sqcup p_2\,\{P_2\}]\ \textcircled{1}\ ; \Gamma_1; \mathcal{L}_1; \mathcal{R}_1 \vdash e : T' \triangleright p'\,\{P'\} \dashv \Gamma'; \mathcal{L}'; \mathcal{R}' \\ \mathit{impure}; pc; \sigma; \Gamma_2; \mathcal{L}_2; \mathcal{R}_2 \vdash e_0\ e_2 : T'' \triangleright p''\,\{P''\} \dashv \Gamma''; \mathcal{L}''; \mathcal{R}'' \\ \lfloor T' \rfloor = \tau_2 \qquad T' \approx T''\ \textcircled{3} \qquad P' \sqcup P'' = P_3 \qquad p_6 = p_1 \sqcup p \\ P_4 = (\text{if } p_6 = \mathsf{clean} \text{ then } P_3 \text{ else } \{\}) \end{array}}{\mathit{impure}; pc; \sigma; \Gamma; \mathcal{L}; \mathcal{R} \vdash e_1\ e_2 : T' \triangleright pc \sqcup p_1 \sqcup p \sqcup p' \sqcup p''\,\{P_4\} \dashv\ \Gamma' \sqcup^* \Gamma''; \mathcal{L}' \sqcup \mathcal{L}''; \mathcal{R}' \sqcup \mathcal{R}''\ \textcircled{4}} \tag{7}$$

Three aspects are noteworthy:

- **Predicate flow** ①. The argument's qualified type, including its guard P_2, is bound in the captured environment as $\sigma_1[x \mapsto T_1 \triangleright pc \sqcup p_2\,\{P_2\}]$. When the body e is then typechecked in this extended environment, any conditional on x can match its guard and trigger de-poisoning. Different call sites supply different arguments with different guards, so the same function body may de-poison at one call site but not another.
- **Signature compatibility** ③. The premise $T' \approx T''$ requires only that the two results have the same shape and surface types; poison status and closure sets may differ, because different closures capture different environments and may produce different poisons or accumulate different closure sets. This relaxation of type equality is sound because the conclusion joins poisons from both alternatives ($p' \sqcup p''$) and merges environments via $\sqcup^*$, so no information is lost.
- **No sequential threading across closures**. Each closure's body is typed independently, from the *same* pre-context (the state after e_2 and before any closure body), never from one already carrying another closure's effects; the per-closure results meet only in the conservative joins of the conclusion ④. No environment update is therefore carried from one closure into the next, and the order in which closures are processed does not affect the result. In Lean this is implemented directly, as a fold over the closure set.

A.3 The Lean Formalisation

Listing 3 gives the Lean statement of the top-level theorem sketched in Figure 6. The listing omits the extern registry to focus on the core language. In the Lean listings, blue names correspond to components of the schematic figures, whose notation Table 3 lists; grey marks additional mechanised hypotheses with no schematic counterpart.

```
1 theorem TypeSystemEnforcesFaultRobustness (e : Expr) (s : InputSpec)
      (obs : ObservableSpec) :
2   wfInputSpec s → wfObsSpec obs →
3   ∀ (ctx : TypeContext),
4     ctxMatchesInputSpec s ctx → wfTypeContext ctx → ctx.pc =
      false → ctx.tagEnv = {} →
```

```
    ∀ (predSat : PredSat) (tm : TagMeaning),
      wfTagMeaningCtx tm ctx → PredSatLaws predSat tm →
      (∃ (fuel : Nat) (res : TypeResult),
        typecheck ctx e fuel = .ok res ∧ ObsConstraints obs res)
      → FaultRobustnessPointwise e s obs ctx predSat tm
```

Listing 3. Main theorem (abridged).

The core statement is: for every expression e and typing context, and hence every regime, if e typechecks (`typecheck`; $ctx \vdash e : Q \dashv ctx'$) and the result type satisfies the observable constraints (`ObsConstraints`; $(Q, ctx') \models obs$), then the two-run fault-robustness relation `FaultRobustnessPointwise` (FR) holds. The remaining hypotheses state that the input and observable specifications must be well-formed (`wfInputSpec`, `wfObsSpec`; $\mathsf{wf}(crit, obs)$), the typing context `ctx` must extend the input specification (`ctxMatchesInputSpec`; $ctx \trianglerighteq crit$) and be well-formed (`wfTypeContext`), and execution starts with a clean program-counter and an empty tag environment. Two additional parameters, `PredSat` and `TagMeaning`, treat predicate satisfaction as an abstract relation. They are universally quantified and constrained by consistency laws (`PredSatLaws`, Section A.4.2) so that the theorem holds for every conforming interpretation.

The theorem relies on the fault-robustness property from Figure 5, whose Lean 4 formalisation is given in Listing 4.

```
def FaultRobustnessPointwise (e : Expr) (s : InputSpec) (obs :
    ObservableSpec)
    (ctx : TypeContext) (predSat : PredSat) (_tm : TagMeaning) : Prop
    :=
  ∀ (μ₁ μ₂ : Memory) (ρ₁ ρ₂ : VarEnv) (θ₁ θ₂ : RealizationMap) (a₁ a₂
    : Address) (pm : Bool),
    ∀ (cf₁ cf₂ : Config) (o₁ o₂ : Value),
      wfRegime ctx.regime s → wfRealizationMap θ₁ →
    wfRealizationMap θ₂ →
      wfConfig { memory := μ₁, varEnv := ρ₁, nextAddr := a₁, pureMode
    := pm } →
      wfConfig { memory := μ₂, varEnv := ρ₂, nextAddr := a₂, pureMode
    := pm } →
      inputsConformSpec s μ₁ ρ₁ θ₁ → inputsConformSpec s μ₂ ρ₂ θ₂ →
      runtimeInstantiatesCtx ctx μ₁ ρ₁ θ₁ → runtimeInstantiatesCtx
    ctx μ₂ ρ₂ θ₂ →
      trustedVarsClosureRegimeInit predSat ctx.regime s ctx μ₁ ρ₁ θ₁
    →
      trustedVarsClosureRegimeInit predSat ctx.regime s ctx μ₂ ρ₂ θ₂
    →
      consistentRegimeSpec predSat ctx.regime s μ₁ ρ₁ θ₁ →
      consistentRegimeSpec predSat ctx.regime s μ₂ ρ₂ θ₂ →
      agreeOnCriticalInputsUnderRegimeSpec predSat ctx.regime s μ₁ ρ₁
    θ₁ μ₂ ρ₂ θ₂ →
      agreeOnTrustedInputs predSat ctx.regime s ctx μ₁ ρ₁ θ₁ μ₂ ρ₂ θ₂
    →
```

Table 3. Notation of Figures 5 and 6 and its Lean 4 counterparts.

Schematic	Lean	Schematic	Lean
$crit$	`InputSpec`	$\langle e, S\rangle \Downarrow \langle o, S'\rangle$	`terminatesTo`
obs	`ObservableSpec`	$(o_1, S_1') =^{obs} (o_2, S_2')$	`observablesEqual`
ctx	`TypeContext`	$\mathsf{FR}(e, crit, obs, ctx)$	`FaultRobustnessPointwise`
$S \models ctx$	`runtimeInstantiatesCtx`	$\mathsf{wf}(crit, obs)$	`wfInputSpec`, `wfObsSpec`
$\mathsf{wf}(ctx.\mathcal{R}, S_1, S_2)$	`consistentRegimeSpec`	$ctx \trianglerighteq crit$	`ctxMatchesInputSpec`
$S_1 \sim^{crit}_{ctx.\mathcal{R}} S_2$	`agreeOnCriticalInputsUnderRegimeSpec`	$ctx \vdash e : Q \dashv ctx'$	`typecheck`
$S_1 \sim^{\overline{crit}}_{ctx} S_2$	`agreeOnTrustedInputs`	$(Q, ctx') \models obs$	`ObsConstraints`

```
16      terminatesTo e { memory := μ₁, varEnv := ρ₁, nextAddr := a₁,
      pureMode := pm } cf₁ o₁ →
17      terminatesTo e { memory := μ₂, varEnv := ρ₂, nextAddr := a₂,
      pureMode := pm } cf₂ o₂ →
18      ∃ (θ₁' θ₂' : RealizationMap),
19        wfExtendsMap θ₁ θ₁' ∧ wfExtendsMap θ₂ θ₂' ∧
20        observablesEqual obs θ₁' θ₂' cf₁ cf₂ o₁ o₂
```

Listing 4. Fault-robustness property in Lean 4.

We highlight only the key structure and omit the extern-registry components of runtime configurations (the full formalisation is included in the development). The property considers two runs of the same expression e under a single typing context but with potentially different runtime states: memories μ_1, μ_2 and variable environments ρ_1, ρ_2—the figure's states S_1, S_2—plus realisation maps θ_1, θ_2, explained below. Table 3 maps the names of Figures 5 and 6 to their Lean counterparts (`wfRegime` keeps its name); the observable outputs are compared up to extended realisation maps that account for freshly allocated locations.

In the formalisation, we also distinguish between two notions of heap location. The type system speaks about abstract *locations*, which appear in points-to sets and typing environments, whereas execution manipulates physical *addresses* in the runtime heap. In a two-run scenario, the physical addresses may differ between the runs while still corresponding to the same abstract location. We use *realisation maps* (θ) to connect the former to the latter, separately for each run.

The guards P of Section 3.1 have the Lean type `Predicate`, with constructors `guardClosure`, `guardOnly`, and `noPred` (the absent guard, written {}); `guardClosure` also stores the captured environment σ.

A.4 Proof Techniques

This section walks through representative techniques used in the proof outlined in Section 5.

A.4.1 Conditional Agreement on Clean Types The fault-robustness property requires two runs to produce agreeing values, but only when the result type is clean under the regime. We capture this through the predicate `resultAgreesIfClean` (Listing 5), used in the proof to compare return values when establishing `observablesEqual`. It is defined as an implication: if the

qualified type is clean under $\mathcal{R}$, then the two result values must agree. For poisoned types `resultAgreesIfClean` is always true because `qtypCleanUnder` is false.

```
inductive qtypCleanUnder (R : Regime) : QTyp → Prop where
  | clean {qt : QTyp} (hpoison : qt.poison = false) :
      qtypCleanUnder R qt
  | guard {qt : QTyp} (k : Tag) (htag : qtypGuardTag qt = some k)
      (hregime : R.lookup? k = some true) (hnotfun :
    qt.etyp.isFunctionType = false) :
      qtypCleanUnder R qt

def resultAgreesIfClean (predSat : PredSat) (R : Regime) (μ₀ μ₁ :
    Memory)
    (θ₀ θ₁ : RealizationMap) (qt : QTyp) (o₀ o₁ : Value) : Prop :=
  qtypCleanUnder R qt →
    agreeValueUnder predSat R μ₀ μ₁ θ₀ θ₁ qt o₀ o₁
```

Listing 5. Conditional agreement predicates.

A qualified type counts as *clean under* $\mathcal{R}$ in one of two ways: its outer poison is literally clean, or it carries a guard tag k with $\mathcal{R}(k) = \mathsf{true}$ (the latter only for non-function types, since guard tags do not de-poison function closures). This two-constructor definition mirrors the two paths to cleanliness in the type system: direct cleanliness and predicate-based de-poisoning.

Value agreement itself is a mutual inductive relation with a second level of conditionality for function types. Two closures agree when they share the same parameter name and body, and their captured environments agree on entries that are clean under $\mathcal{R}$. When the function's *internal* arrow poison is dirty, agreement holds trivially regardless of captured environments. This mirrors the outer-level structure: poisoned components impose no constraints, and only clean components must match between runs.

A.4.2 Abstract Predicate Satisfaction The fault-robustness proof reasons about both semantic execution (big-step evaluation) and typing derivations. Guard predicates $y \Rightarrow pe$ sit at the boundary: the typing side captures the *typing* environment σ, while the semantic side needs to know whether the predicate holds for a runtime value. Bridging the two directly would require evaluating the predicate body pe with a concrete closure over runtime bindings, which is unavailable in the proof context.

We resolve this tension by treating predicate satisfaction as an abstract relation $\mathsf{PredSat} : \mathit{Tag} \to \mathit{Memory} \to \mathit{Value} \to \mathsf{Prop}$ that specifies whether the predicate identified by tag k holds for a value in a given memory state, without requiring concrete evaluation. The captured environment is encoded in the tag's identity: each tag uniquely identifies both the predicate body and its lexical context. The fault-robustness theorem quantifies over *all* `PredSat` relations satisfying `PredSatLaws`, making the proof parametric in what predicates mean. `PredSatLaws` requires two properties:

- **Bidirectional matching** (`match_iff`). If `matchGuard` recognises a condition as checking tag k and the condition evaluates to b at runtime, then $b = \mathsf{true}$ if and only if $\mathsf{PredSat}(k, \mu, v)$ holds for checked value v. This lets the proof convert a successful guard test into a predicate fact—the proof-level counterpart of the `match_guard`-based de-poisoning of Section 3.3.
- **Memory monotonicity** (`mem_monotone`). If two memory states agree on all locations in the predicate's declared dependency set, predicate satisfaction is preserved. This lets the proof transport predicate facts across heap updates that do not affect the guard, mirroring the dependency-set invalidation of $\mathcal{L}$ and $\mathcal{R}$ (Rule 5).

A.4.3 Carry-over Invariants The proof proceeds by induction on expression structure, which requires several auxiliary invariants, themselves proven by induction on the same structure and mutually dependent. Rather than proving each invariant as a separate lemma, we bundle them into a single strengthened *step property* (`StepFR`) that operates on one expression with a pre-context and a post-context. `StepFR` assumes a set of invariants as premises and re-establishes them in its conclusion for the updated context after evaluation. This design lets the inductive proof chain steps together: the post-conditions of one step become the pre-conditions of the next.

The main invariants carried through each step are:

- **Runtime–typing agreement.** Each run's memory and variable environment match the typing context (values inhabit their declared types, realisation maps are consistent).
- **Regime consistency.** $\mathcal{R}$ correctly reflects predicate satisfaction for each run: when $\mathcal{R}(k)$ holds, the predicate tagged k holds for the corresponding runtime value and vice versa.
- **Two-run agreement on clean entries.** The two runs agree on every variable and memory location whose type is clean under the current regime.

The full `StepFR` statement also carries a number of auxiliary well-formedness and side conditions (for example tag-meaning well-formedness, realisation-map extension, and value-level regime consistency); we omit these here for brevity. The step theorem additionally concludes that the two result values agree whenever the result type is clean, which is the per-expression analog of fault robustness. The top-level theorem instantiates `StepFR` with the initial context derived from the input specification and extracts the observable-output agreement from the final post-context.

Some of these invariants are *deep*: they must hold not only for the top-level typing context but also recursively for environments captured inside closures. For example, regime consistency requires that predicate satisfaction agrees with the regime for values bound in every transitively captured environment, not just the immediately visible bindings.

Because the step property already re-establishes runtime–typing agreement and well-formedness for the post-context, we believe a standard type-preservation

theorem could be derived from `StepFR` as a corollary, though we have not formalised this separately.

A.4.4 Function Application Proof Function application is the largest single proof component, comprising approximately 21.7 KLOC and 115 lemmas (theorem and lemma declarations). The complexity arises from the interaction of the mechanisms described in Section 3.5, all of which must be verified simultaneously.

The proof begins by establishing a *body-typechecking pre-condition* property: when a closure (x, e, σ_{cap}) is applied to an argument, the body e is retypechecked in the captured environment extended with the argument's qualified type (Rule 6 ①). The proof must show that all carry-over invariants (runtime–typing agreement, regime consistency, two-run agreement on clean entries) transfer from the caller's context into this reconstructed body context. In particular, predicate flow ① means the argument's guard enters the body's typing environment, so the proof must also establish that the argument's regime consistency and value agreement carry over into the extended captured environment. Because closures capture their environment at definition time, this requires *deep* invariant threading: the captured environment's well-formedness and regime consistency must be maintained across every intervening evaluation step since closure creation.

When a conditional merges closures from different branches into a single closure set (Rule 7), the proof must handle each alternative and combine results via *join-transport* lemmas. These lemmas show that when signature compatibility ③ holds across alternatives, carry-over invariants established for one alternative can be transported to the joined result type. The transport is non-trivial because the joined result may differ in poison, guard, and closure set from the alternatives: $T' \approx T''$ ③ permits different poisons and closure sets, and the conclusion joins them via $\sqcup^*$.

A.4.5 Ghost Location Handling In the fault-robustness proof, each of the two runs maintains its own realisation map θ (Section 4.1), connecting abstract locations in Γ to physical heap addresses. As evaluation allocates new references, the map grows; the proof must show at each step that the extended map preserves existing bindings (`wfExtendsMap`). The two maps are then combined when establishing value agreement: for a reference typed at location set LS (Section 3.4), the proof looks up $\theta_1(l)$ and $\theta_2(l)$ independently and compares the values stored at the resulting addresses.

A subtlety arises from *ghost locations*, which are abstract locations that appear in Γ but are never allocated at runtime. This occurs when a location is introduced by the typing context but the corresponding allocation is unreachable in a particular execution. The agreement predicate `agreeLocUnder` handles this with two constructors: either both runs realise the location (and the stored values must agree), or both leave it unrealised. This ensures that ghost locations

satisfy the specification trivially without requiring the proof to track which locations are live.

A.4.6 Conditional Branch Combinations For compound expressions, the two-run fault-robustness argument proceeds by applying the inductive hypothesis to the *same* sub-expression in both runs. Conditionals break this pattern: when the two runs take different branches, one run evaluates the then-branch while the other evaluates the else-branch, and the proof must relate results of two unrelated expressions. This proof structure follows the typing treatment of conditionals from Section 3.3. The proof splits into three cases based on the status of the condition:

1. **Fixed outcome.** When the regime or the logical environment fixes the guard outcome (Rule 1, Section 3.3), both runs are guaranteed to take the same *known* branch. The proof applies the inductive hypothesis to that branch directly.
2. **Deterministic condition.** When the condition is clean, both runs agree on its value by the carry-over invariant, so they take the same branch, but the proof does not know which one. The proof therefore considers both branches and joins the resulting post-environments.
3. **Non-deterministic condition.** When the condition is poisoned, the two runs may evaluate it to different booleans, producing results from different branches. The two-run inductive hypothesis is then no longer applicable, because the two runs are evaluating different expressions, so the proof switches to single-run preservation for each taken branch. This suffices because the branch choice itself makes the result type poisoned (Rule 3), so no cross-run agreement obligation remains.

In the latter two cases, the typing rule joins the flow-sensitive outputs of the two branches. The ordinary joins on $\mathcal{L}'$ and $\mathcal{R}'$ retain only facts that agree between the branches, preventing branch-local logical or regime information from leaking into the post-context. The typing environment Γ' is joined via $\sqcup^*$ (strong join), which strips guards from entries that differ between branches and thereby avoids guard conflicts in the post-context.