Soteria: Efficient Symbolic Execution as a Functional Library

Perhaps You *Should* Write Your Own Symbolic Execution Engine!

SACHA-ÉLIE AYOUN, Imperial College London, United Kingdom and Soteria Tools Ltd., United Kingdom
OPALE SJÖSTEDT, Imperial College London, United Kingdom and Soteria Tools Ltd., United Kingdom
AZALEA RAAD, Imperial College London, United Kingdom and Soteria Tools Ltd., United Kingdom

Symbolic execution (SE) tools often rely on intermediate languages (ILs) to support multiple programming languages, promising reusability and efficiency. In practice, this approach introduces trade-offs between performance, accuracy, and language feature support. We argue that building SE engines *directly* for each source language is both simpler and more effective. We present Soteria, a lightweight OCaml library for writing SE engines in a functional style, without compromising on performance, accuracy or feature support. Soteria enables developers to construct SE engines that operate directly over source-language semantics, offering *configurability*, compositional reasoning, and ease of implementation. Using Soteria, we develop Soteria$^{\text{Rust}}$, the *first* Rust SE engine supporting Tree Borrows (the intricate aliasing model of Rust), and Soteria$^{\text{C}}$, a compositional SE engine for C. Both tools are competitive with or outperform state-of-the-art tools such as Kani, Pulse, CBMC and Gillian-C in performance and the number of bugs detected. We formalise the theoretical foundations of Soteria and prove its soundness, demonstrating that sound, efficient, accurate, and expressive SE can be achieved without the compromises of ILs.

CCS Concepts: • **Theory of computation** → **Automated reasoning**; *Verification by model checking*; **Program analysis**.

Additional Key Words and Phrases: Symbolic Execution, Rust, Functional Programming, Incorrectness Logic

ACM Reference Format:
Sacha-Élie Ayoun, Opale Sjöstedt, and Azalea Raad. 2026. Soteria: Efficient Symbolic Execution as a Functional Library: Perhaps You *Should* Write Your Own Symbolic Execution Engine!. *Proc. ACM Program. Lang.* 10, PLDI, Article 228 (June 2026), 26 pages. https://doi.org/10.1145/3808306

1 Introduction

In the last two decades, a number of symbolic execution (SE) tools were designed with the intent of supporting *multiple languages*, including Infer [11, 32] for Java, C, Hack, and Python, Gillian [22, 38] for JavaScript, C, and Rust [4], as well as others [1, 10, 42]. To do this, each such tool relies on an *intermediate language* (IL) and offers an SE engine that can execute IL code. A putative advantage of having an IL is that one can avoid re-implementing an SE engine for each new language L: instead, one can obtain it by *compiling* L to IL. In this work, we argue that one *should* implement an SE engine for each language and start by addressing two arguments commonly made in favour of ILs.

Authors' Contact Information: Sacha-Élie Ayoun, Imperial College London, London, United Kingdom and Soteria Tools Ltd., London, United Kingdom, s.ayoun17@imperial.ac.uk; Opale Sjöstedt, Imperial College London, London, United Kingdom and Soteria Tools Ltd., London, United Kingdom, opale.sjostedt23@imperial.ac.uk; Azalea Raad, Imperial College London, London, United Kingdom and Soteria Tools Ltd., London, United Kingdom, azalea.raad@imperial.ac.uk.

ACM 2475-1421/2026/6-ART228
https://doi.org/10.1145/3808306

Argument 1: Design once, use often! This oft-touted mantra of ILs refers to their reusability when adding support for a new language. In practice, however, designing both an efficient and sufficiently expressive IL is far from straightforward, and it requires *foresight* to predict all possible features in future languages. In our experience, this then results in either a bloated IL that is *inefficient*, or an *incomplete* IL that cannot support the desired features. In the latter case, one can either revisit the IL to extend it with the required features, defeating the goal of 'design once'; forgo supporting these features altogether, resulting in incomplete reasoning; or extend the compiler with complex pre-analyses to offset the lack of support in the IL, resulting in *ad hoc solutions*.

Practically all three of these IL-related issues—inefficiency, incompleteness, and ad hoc solutions—are illustrated in the various attempts of existing tools to analyse Rust using ILs, such as CBMC's GOTO [30] (used by Kani [57]) or GIL (used by Gillian-Rust [4]). For example, GOTO does not support detecting uninitialised memory accesses, which are undefined behaviours (UBs), and so the Kani compiler performs an entire points-to analysis to enable this detection, which still remains incomplete. Further, detecting aliasing violations as per the intricate Tree Borrows [58] aliasing model of Rust remains entirely beyond the reach of Kani. On the other hand, Gillian-Rust encodes Rust pointers and mutable borrows in creative ways because GIL does not have the constructs to express these concepts naturally, hindering the expressivity and efficiency of the analysis.

Note that although using an *existing* IL (e.g. LLVM-IR or Wasm) eschews the need for a dedicated compiler, it often leads to *inaccurate* analyses. This is because existing compilers are not *semantics-preserving* [55] and erase UBs and other information deemed irrelevant to the IL. As such, analysing the resulting IL code may lead to *false negatives*. For instance, Wasm does not encode any information required for reasoning about Tree Borrows, and hence Owi [1], an SE engine with Wasm as its IL, cannot find aliasing bugs. Moreover, compilation causes Owi's IL code to grow substantially in size, which is likely to make its Rust analysis less efficient. All of this leads us to our first claim:

Claim 1: *IL-design is a trade-off between performance, accuracy, and comprehensive support for language features, and one cannot maximise all aspects of this trio at once. A dedicated analysis can.*

Argument 2: Prove once, use often! From the meta-theoretical point of view, an IL-based SE engine can be proven sound against its IL once and for all, and then all existing and future analyses built over the IL should inherit its soundness guarantees. This, however, makes a *significant assumption*, namely that the source-to-IL compiler is also sound. Not only is this often not the case in practice [27], verifying (proving sound) a compiler designed for analysis is a highly challenging task [44]. Moreover, as discussed above, compilers often erase UBs and other information unsupported or deemed irrelevant by the IL, leading to gaps in expressivity or even *false negatives* in the analysis (e.g. such false negatives occur in Gillian-C, see §5). This brings us to our second claim:

Claim 2: *A trustworthy interpreter for* L *is not harder to write than a trustworthy* L*-to-*IL *compiler.*

The Soteria Framework. Motivated by our two claims above, we present *Soteria*, a simple, yet powerful, OCaml library for writing SE engines. A key novelty of Soteria is that it abstracts the fixed-IL aspect of the analysis, allowing one to construct an SE engine that operates *directly* on the source language or on an IL close to the source-language semantics. Soteria provides a *monadic interface*, together with intuitive syntactic sugar, to write SE engines in functional style. To show the utility of Soteria, we build two SE engines over it: Soteria$^{\text{Rust}}$, our automated symbolic testing engine for Rust; and Soteria$^{\text{C}}$, our automated, compositional, bi-abduction engine for C (see below). Using these two case studies, we show the advantages of Soteria along several dimensions.

First, we show that Soteria yields *highly performant* engines comparable or better than the state of the art. For example, Soteria$^{\text{Rust}}$ competes in performance with Kani (a Rust SE engine maintained by a team of AWS engineers over the last four years) [57], while covering a significantly

larger subset of the Rust semantics (see §4). To our knowledge, Soteria$^{\text{Rust}}$ is the *first* SE engine to support detecting bugs related to the Tree Borrows aliasing model [58], while *no existing verification tool* [3, 4, 19, 21, 24, 26, 31, 33] supports Tree Borrows. Notably, using Soteria$^{\text{Rust}}$ we detected and reported a potential bug in `hashbrown` [48], the *second-most-downloaded* Rust crate [49] (see §4).

Second, we show that building SE engines over Soteria is simple: as presented here, Soteria$^{\text{Rust}}$ took a first-year PhD student *merely eight months* to implement. This is partly due to the simple monadic interface of Soteria that allows one to write SE engines in a functional style that closely resembles a concrete interpreter for the source language, and partly because Soteria provides a library of *reusable components* that can be used across engines. Indeed, the student re-used many of the components already developed for Soteria$^{\text{C}}$ in Soteria$^{\text{Rust}}$.

Finally, we formalise the soundness guarantees of Soteria and the engines built over it. Specifically, following previous work [5], we *formalise* the theoretical foundation of Soteria and prove that the symbolic interpreters built over it are sound as long as they meet certain conditions (§6).

We further highlight that Soteria is *highly flexible and configurable* in three ways. First is the choice of the *source language*: clients of Soteria can build analyses over it for any language L, so long as they develop an interpreter for L. More notably, they inherit the *soundness guarantees* of Soteria by construction, provided they meet the conditions of §6.

Second, Soteria is designed to come with *batteries included* and provides out-of-the-box support for logging and statistics. Indeed, logs in Soteria are often more informative than those in IL-based frameworks, as they relate directly to the relevant source code rather than to the IL.

Third, inspired by Lööw et al. [35], Soteria can be *configured* to perform analysis in one of two *modes*: *under-approximate* (UX) analysis for *bug catching* [43, 46] (as in Pulse [32]), or *over-approximate* (OX) analysis for *bounded verification* (as in e.g. CBMC). Specifically, the analysis mode *m* is passed to the *SE monad* at the core of Soteria, which determines the execution path exploration strategy à la *m* (exploring *all* paths in OX and *some* paths up to a depth in UX). Importantly, our monadic approach is also compatible with *unbounded* verification using compositional SE [5, 28].

Lastly, Soteria is flexible enough to implement various analyses as long as they can be expressed as symbolic execution. Bounded whole-program symbolic testing (WPST) is one such analysis (as in CBMC [30]); bug-finding by means of under-approximate *bi-abduction* (as in Pulse [32]) is another (as per [23, 35, 36]). For instance, Soteria$^{\text{C}}$ supports *both* WPST and fully automatic bi-abductive bug detection. As we show in §5, WPST in Soteria$^{\text{C}}$ is competitive in performance with CBMC. Moreover, its compositional capabilities are competitive in performance with the industry-grade Infer.Pulse (henceforth Pulse) tool, while being *an order of magnitude faster* than Gillian-C [38].

Contributions and Outline. Our contributions, described intuitively in §2, are as follows.

- (§3) We present Soteria, a functional library for building *automated, industry-scale* SE engines.
- (§4) We describe Soteria$^{\text{Rust}}$, our Rust SE engine that outperforms the state of the art in bug finding capabilities, and is the *first* SE engine to support Tree Borrows.
- (§5) We present Soteria$^{\text{C}}$, our SE engine for C that supports *both* WPST *and* fully automated bug detection using bi-abduction. Soteria$^{\text{C}}$ is comparable to the industry-grade Pulse tool and outperforms Gillian-C (factor of 2) and CBMC (by an order of magnitude).
- (§6) We formalise the theory behind Soteria and prove the soundness of its guarantees.
- (§7) We discuss key design choices we made in Soteria and their limitations.
- (§8) We conclude with a discussion of related work.

2 Overview: Build Your Own Symbolic Execution Engine for Lang over Soteria

We depict an overview of the Soteria architecture in Fig. 1. Soteria is an *efficient*, functional programming library written in OCaml that provides:

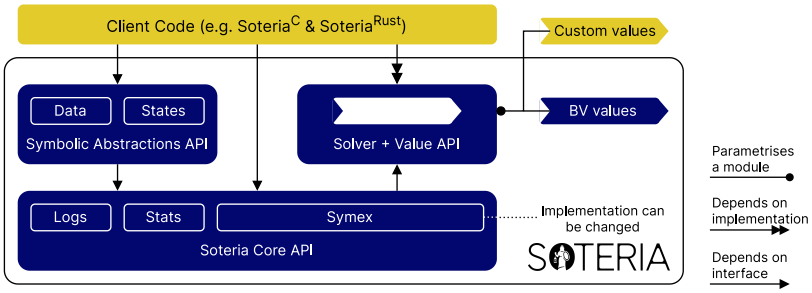


Fig. 1. Overview of the Soteria architecture

- a monadic interface with all required primitives to write a symbolic execution (SE) engine;
- an efficient implementation of this interface, parametrised on a user-defined value language;
- a pre-defined instantiation of the value language for convenience;
- pragmatic, developer-friendly tooling for aggregating statistics and logs; and
- ready-made data structures for SE that can be soundly used within symbolic computations.

We present a didactic overview of Soteria as a tutorial on using it to implement a *symbolic interpreter*. We do this for a simple side-effect-free expression language, Lang. Our aim is not to give an exhaustive account of Soteria's features, but rather an intuitive account of using it. For lack of space we elide the details unnecessary for understanding the intuition behind Soteria. Nevertheless, we present the full details in the extended version [59]. The Lang syntax is standard and comprises the expected constructs of let-binding, if-else branching, and so forth.

Preliminiary: Custom Let-Operators in OCaml. Soteria makes extensive use of OCaml's *custom let-operators*, which (though daunting at first glance) are simply syntactic sugar for a monadic `bind` operator. Specifically, given a monad `M`, one can define `let ( let* )= M.bind`, whereafter OCaml will desugar each subsequent occurrence of `let* x = e1 in e2` to `M.bind e1 (fun x → e2)`.

Preliminary: Concrete Monadic Interpreter. In the snippet across we define a simple *concrete monadic interpreter* for the `If` construct of Lang. We define the interpreter using OCaml syntax and introduce concepts required to later understand Soteria. First, we assign the monadic bind operator of the `ExecutionMonad` (defined in the extended version [59]) to the `let*` operator. Effectively, `let* x = e1 in e2` executes the monadic computation `e1`, which may be non-deterministic and may error (i.e. it returns a set of results that are either values or errors). Each error result of `e1` is propagated to the set of final results; for each successful (non-error) result v, it continues by executing `e2` with `x` bound to v.

```
let ( let* ) = ExecutionMonad.bind
let eval subst expr =
match expr with
| If (cond, then_e, else_e) →
  let* cond_v = eval subst cond in
  let* cond_b = bool_of_val cond_v in
  if cond_b then eval subst then_e
            else eval subst else_e
```

When interpreting the `If` construct, we first evaluate the guard `cond`; once again it may error and yield multiple values. For each successful result `cond_v`, we convert it to an OCaml boolean `cond_b` using `bool_of_val`, which may also error if `cond_v` is not a boolean (i.e. if the evaluated program is ill-typed). Finally, if `cond_b` is true, we evaluate expression `then_e`; otherwise, we evaluate `else_e`.

Note that this interpreter, while written in OCaml *syntax*, might not be computable: evaluating an expression may result in infinitely many results, e.g. evaluating `NondetInt` can yield *any* integer. For instance, the program below yields the (infinite) set of results $\{(ok, x) \mid x > 5\} \cup \{(err, x) \mid x \leq 5\}$.

```
Let x = NondetInt in (If (x > 5) then Ok x else Error x)
```

Symbolic Interpreter with Soteria. Lifting our concrete interpreter above to a *symbolic* one is straightforward, as shown across. It is almost identical to the concrete interpreter, except that (1) `let*` is now bound to the `Symex.Result.bind` operator, sequencing *symbolic computations*; and (2) instead of branching on a concrete boolean using `if`, we branch on a *symbolic* one using `if%sat`.

```
let ( let* ) = Symex.Result.bind
let eval subst expr =
match expr with
| If (cond, then_e, else_e) →
  let* cond_v = eval subst cond in
  let* cond_b = bool_of_val cond_v in
  if%sat cond_b then eval subst then_e
                else eval subst else_e
```

A symbolic computation is a computation that depends on *symbolic variables*, which represent unknown values. A symbolic computation yields a set of results; each result is associated with a *path condition*, a symbolic boolean (boolean expression with symbolic variables) constraining the symbolic variables under which the result occurs. For instance, consider the example above. During SE, `NondetInt` will yield a *single branch* where `x` is bound to a fresh symbolic variable $\widehat{x}$ representing an unknown integer, with the path condition true (i.e. with no constraint on $\widehat{x}$). Indeed, this is a key advantage of SE: infinitely many branches of concrete execution can be represented by a finite number of symbolic branches with symbolic variables. The `Symex.bind` operator sequences symbolic computations by simply threading the path conditions appropriately within each branch.

The `if%sat` construct (which is syntactic sugar for the Soteria `Symex.branch_on` primitive – see §3), lifts the concrete notion of conditional branching to the symbolic world. Its condition is a *symbolic* boolean (rather than a concrete one), i.e. a boolean expression that may depend on symbolic variables. It then behaves as expected for SE. First it checks if the guard ($\widehat{x} > 5$ in the example above) is satisfiable; if so, then it executes the `then` branch. Next, it also checks if the negation of the guard ($\widehat{x} \leq 5$ in the example above) is satisfiable; if so, then executes the `else` branch. As such, the SE of the example above returns two branches: (1) $\langle ok : \widehat{x} \mid \widehat{x} > 5 \rangle$, the (then) branch resulting in the successful result $ok : \widehat{x}$ with path condition $\widehat{x} > 5$; (2) $\langle err : \widehat{x} \mid \widehat{x} \leq 5 \rangle$, the (else) branch resulting in the error result $err : \widehat{x}$ with path condition $\widehat{x} \leq 5$.

The Soteria Symex Module. The symbolic interpreter described above is implemented using our `Symex` module in Soteria. In addition to the `let*` and `if%sat`, the `Symex` module provides a number of primitives that enable users to write *efficient* symbolic interpreters from scratch with *minimal effort*. These primitives include e.g. `nondet` for introducing fresh symbolic variables, or `branches` for creating multiple branches from a list of computations. The latter is useful for modelling real non-determinism (e.g. allocation that may succeed or fail). Finally, `Symex` provides the `run` function that receives three arguments: (1) a symbolic computation; (2) an 'execution mode' that determines whether the analysis is *over-approximate* (OX) or *under-approximate* (UX) (inspired by [35]); and (3) an optional *fuel* to limit the breadth and depth of execution (infinite by default). It then executes the computation and returns a list of results, each paired with its path condition.

Soteria provides pre-defined common data structures (`Data` module) and state monad transformers (`State` module) that can be soundly used within symbolic computations, enabling code sharing between various interpreters. Through this reusability and its `Symex` module, Soteria simplifies the task of developing *efficient* SE engines. For instance, as we show in §4, this enabled a *first-year* PhD student to develop a highly efficient Rust SE engine that is competitive in performance to the state-of-the-art tool Kani [57] by AWS (developed and maintained by *seasoned industry engineers* in the last *four years*), while covering a substantially larger subset of Rust, including Tree Borrows [58].

Symbolic Values and Solvers. Soteria is *highly general* in that users can obtain a *bespoke* `Symex` *module tailored* to their needs. Specifically, Soteria provides a *functor*, `Soteria.Symex.Make`: given a module implementing the Soteria `Solver` interface, it produces a `Symex` module tailored to the symbolic value language that the solver can reason about. In the case of Lang above, symbolic

values need only to range over integers and booleans; as such, the symbolic value language supplied to `Soteria.Symex.Make` only needs to implement these two sorts, as well as operations that may cause branching, e.g. integer comparison. Given such a solver, let us call it `Simple_solver`, the `Symex` module above can be simply obtained by a *single line*: `module Symex = Soteria.Symex.Make (Simple_solver)`.

In practice, users of Soteria may often want to re-use an existing solver implementation rather than implementing their own from scratch. To this end, Soteria comes with a pre-defined solver module `BV_solver` (described later in §3.1) that implements a symbolic value language based on bit-vectors, comes with several important *optimisations*, and uses Z3 as its underlying SMT solver. That is, while Soteria affords its users the *flexibility* to supply their own solvers, it also comes packaged with *efficient, general purpose and ready-to-use* components out of the box.

Guarantees. The Soteria `Symex` monad provides important soundness guarantees, summarised as follows. (1) Primitives of the `Symex` monad are sound against their concrete counterparts; e.g. `Symex.nondet` indeed abstracts over concrete non-determinism; (2) sequential composition of sound symbolic computations is sound against the sequential composition of their concrete counterparts; (3) the symbolic `if%sat` construct is sound against the concrete `if` construct. While the user must ensure they compose sound primitives that faithfully capture the concrete semantics of their interpreted language, we believe that doing so is no more difficult than writing a *correct* compiler.

Finally, Soteria provides out-of-the-box support for logging and statistics. Logging in Soteria is directly integrated in SE: users can log messages at any point of execution, and Soteria can create an HTML report that groups logs by execution path (see extended version [59] for an example).

3 Implementation

Recall from §2 that Soteria comprises several components as depicted in Fig. 1. In what follows, we detail the implementation of the `Solver`, `Symex`, and `Data` modules. We highlight several *optimisations* we have implemented in Soteria, and we signpost them clearly with labels **Oi**. These optimisations, while minor in isolation, have been carefully designed to work in synergy and have a significant impact on the overall performance of symbolic execution.

3.1 Solver and Values

Recall that symbolic execution (SE) uses symbolic variables to abstract over sets of values. As discussed in §2, Soteria is parametric in its choice of symbolic variables. Specifically, instantiating symbolic variables in Soteria requires implementing the `Value` and `Solver` modules. The `Value` module describes the carrying type of symbolic values (or symbolic expressions), while the `Solver` module describes a stateful object to which constraints (i.e. boolean symbolic values) can be added, and which has a `check_sat` function checking whether the current set of constraints is satisfiable.

These two modules have clearly defined interfaces, and the arrows in Fig. 1 indicate that `Symex` depends on the *interfaces* of `Solver` and `Value`, while clients of Soteria may additionally depend on the *implementation* of `Value`, which is necessary to construct symbolic values in the first place.

The Value Interface. The value interface (slightly simplified here for clarity) exposes a minimal set of operations and types required to manipulate symbolic values during SE. It exposes two types: (1) `t`, carrying values (a popular OCaml convention is to name the main type of a module as '`t`'), and (2) `ty`, carrying physical representations of the type of symbolic values. The interface also exposes `mk_var : Var_id.t → ty → t` to create symbolic variables from a variable identifier and a type; `not : t → t` to negate symbolic booleans, necessary for defining `if%sat` where both the guard and its negation must be checked for satisfiability; and `as_bool : t → bool option` to concretise a symbolic value as a concrete boolean where possible (and otherwise returning `None`). The interface also contains operations for pretty-printing and variable substitution (omitted for brevity).

The Solver Interface. The solver interface defines a 'solver' of type `t` as an object to which constraints (boolean symbolic values) can be added using `add_constraints`. One can check whether the current state satisfies these constraints using `check_sat`, which returns SAT, UNSAT or UNKNOWN (the `Solver_result.t` type). The current state also records the symbolic variables currently in scope, allowing to create *fresh* variables using `fresh_var`. A solver must be *incremental*: it must support saving the current state using `save`, backtracking the last `n` checkpoints using `backtrack_n`, and resetting to the initial state using `reset`. The solver must provide a way to copy the current state of the solver as a list of boolean symbolic values using `as_values`. Finally, it must provide the `simplify` procedure to simplify a given symbolic value according to the current set of constraints; this is particularly important to enable optimisations in the SE monad, as we discuss below.

```
module type Solver = sig
  module Value : Value.S
  type t
  val add_constraints : t → Value.(sbool t) list → unit
  val check_sat : t → Solver_result.t
  val fresh_var : t → 'a Value.ty → Var_id.t
  val save : t → unit
  val backtrack_n : t → int → unit
  val reset : t → unit
  val as_values : t → 'a Value.t list
  val simplify : t → 'a Value.t → 'a Value.t
end
```

Note that this interface is already implemented by SMT solvers such as Z3 or CVC5, which support incremental solving via push/pop commands. As such, one can simply wrap such a solver directly behind this interface and use the solver expression language as the symbolic value representation. However, by keeping the `Value` opaque, we allow for arbitrary abstract implementations of symbolic values, which may enable simplifications that are more tailored to the SE use case. Similarly, using the `Solver` interface directly with an SMT solver is inefficient in practice, as SMT solvers are often designed to solve large batches of constraints at once, rather than incrementally adding small constraints. These two insights motivated the design of the `BV_solver` implementation.

BV_values. The `Bv_Value` module *implements* the `Value` interface, using bitvectors to represent most symbolic values. Specifically, a symbolic value's type is either a bitvector of a given size, an IEEE float of a given precision, a location (memory object identifier) represented as a bitvector, a pointer represented as a location-offset pair (both bitvectors) à la CompCert [34], or a boolean. This is sufficient to represent all values required to symbolically execute C and Rust programs.

Our first optimisation (**O1**) is to hashcons [20] all values to reduce memory consumption, speed up syntactic equality checks, and enable efficient maps and sets of symbolic values. Our second optimisation (**O2**) consists in hiding the data representation of symbolic values to force clients to use constructors that eagerly perform simplifications, following work from Correnson [13]. For instance, constructing the addition of two concrete bitvectors immediately returns their sum as a concrete bitvector. Note that **O1** and **O2** synergise: simplifications ensure that certain values cannot be constructed in the first place, normalising them to a canonical form, hence increasing memory efficiency of the hashconsing table by reducing the number of distinct nodes.

BV_solver. The `BV_solver` module implements the `Solver` interface using a pipeline composed of lightweight analyses inspired by abstract interpretation, manually optimised state management, and an underlying connection to the Z3 SMT solver.

```
type t = {
  var_counter: Var_counter.t;
  analyses: Analysis.t;
  state: Solver_state.t;
  z3_conn: Z3_conn.t }
```

The solver uses a variable counter to track which symbolic variables are currently in scope and to create fresh ones when requested. In our next optimisation (**O3**), when constraints are added using `add_constraints`, they are first passed to `Analysis.t`, which comprises a few *lightweight analyses* (inspired by abstract domains) to store a subset of the current set of constraints *efficiently*. For instance, our *interval analysis* tracks upper and lower bounds on bitvector variables, allowing us to detect unsatisfiable constraints (e.g. `0 < x && x < 0`) quickly without calling the SMT solver, or to simplify constraints such as `0 <= x <= 0` to `x = 0`. Our *equality analysis* tracks equalities between expressions and uses a cost function to then decide which expression to use as a canonical representative when simplifying constraints.

After going through the analyses, constraints that have not been fully stored are added to the `Solver_state.t`, which is a vector of constraints. Saving and backtracking the solver state is implemented by simply recording the size of this vector at each checkpoint.

```
let process =
  let* () = Symex.assume c1 in
  let* () = Symex.assume c2 in
  if%sat c3 then ...
```

Further, in our next optimisation (**O4**), each constraint in the solver state is marked as 'checked' or 'unchecked'. When `check_sat` is called, all unchecked constraints, together with all constraints that share variables with them, are sent to Z3 for checking; if Z3 returns SAT, then all constraints are marked as checked. This avoids sending the *entire* set of constraints to Z3 at each call to `check_sat`, which would be very inefficient in practice. **O4** allows us to add several constraints to the path condition using `assume` *without* incurring the cost of a check when they are added. To see this, consider the following code snippet on the right.

The two calls to `assume` simply add `c1` and `c2` to the solver state as unchecked constraints. When reaching the `if%sat`, a checkpoint is created, `c3` is added to the solver state and all three constraints are sent to the solver to check for satisfiability. If the result is satisfiable, all three constraints are marked as checked. When backtracking to the checkpoint, `c3` is removed and `not c3` is added to the solver state. Since `c1` and `c2` are already marked as checked, only `not c3` and constraints sharing variables with it need to be sent to the solver, which may save us from checking `c1` and `c2` again.

Additionally, as part of our next optimisation (**O5**), we *cache* all queries previously sent to Z3; in our Collections-C case study (§5), this reduces Z3 calls by a factor of 3.8 and execution time by a factor of 3. This further synergises with **O1**, since implementing the cache becomes cheap. It also synergises well with **O2** as simplifications imply a higher probability of cache hits. In fact, we also add construction-time transformations of symbolic values that aim to optimise caching more than solving time, e.g. by always choosing the same ordering of operands in commutative operations.

Finally, our last optimisation (**O6**) uses a `simplify` function which leverages analyses, as well as the solver state, to perform *contextual* simplifications. For instance, a constraint that is already part of the solver state is simplified to `true`, and its negation is simplified to `false`. Surprisingly, such simple optimisations allow for significant reductions in calls to the SMT solver in practice.

We have also developed a lightweight implementation of the Solver interface without **O1**–**O6**. Our lightweight implementation is directly wrapped around Z3, using its push/pop functionality to manage the path condition. However, we found this lightweight approach to be orders of magnitude slower than `BV_solver` empirically. We leave a more in-depth comparison as future work.

3.2 The Symex Module

The `Symex` module provides the core primitives required to define SE. We write a clear interface for `Symex` so that different implementations of the SE monad can be swapped easily. For instance,

one could design monads that explore branches in different orders (e.g. depth- versus breadth-first exploration). We present the Soteria `Symex` interface and our implementation of it.

The Symex Interface. The `Symex` interface exposes only core primitives required for efficient SE. Specifically, it exposes (1) `return` and `bind`, the monadic operations for sequencing SE steps; (2) `branch_on`, the desugared function underpinning `if%sat`; (3) `branches : 'a t list → 'a t`, to explore a list of branches without conditions, required to model non-determinism, e.g. allocation, which may return either a valid allocated object or a null pointer, and `val vanish: unit → 'a Symex.t`, which stops exploration of the current branch; (4) `val nondet : Value.ty → Value.t t`, to instantiate an unconstrained symbolic value for a given type; and (5) `run`, to run an SE monad to completion, returning a list of pairs, each comprising a final result and the path condition (as a list of constraints) leading to this result. As shown below, the `run` function also takes an optional *fuel* parameter, infinite by default, to limit the number of branches or steps explored during SE, as well as a *mode* parameter to choose between over- (OX) and under-approximate (UX) SE.

```
val run:?fuel:Fuel.t → mode:Approx.t → 'a t → ('a * Value.(sbool t)) list
```

While these are the only core primitives *required* to define any symbolic computation, the `Symex` interface exposes additional operations either for convenience, or because they allow for more efficient implementations. These include `assume`, which adds a constraint to the path condition, and `assert`, which checks if a constraint holds and stops exploring the current branch if it does not. These two operations can be implemented using `if%sat` as follows:

```
let assume cond = if%sat cond then return () else vanish ()
let assert cond = if%sat cond then ok () else error AssertError
```

However, providing them as primitive operations allows for more efficient implementations that avoid exploring unnecessary branches. For instance, as discussed above, `assume` may simply add the constraint to the solver state without calling the solver immediately. It would then be the responsibility of the `run` function to check the satisfiability of the path condition when reaching a leaf of the SE tree, if not all constraints have been checked yet.

The Symex Implementation. The `Symex` monad should model non-determinism due to branching and attach a state, the path condition, to each branch of the execution.

```
let state = ref (Solver.create ())
type 'a t = ('a → unit) → unit
      (* = 'a Iter.t *)
```

In practice, there are several ways of implementing both non-determinism and state in OCaml. We make two design choices to improve efficiency. First, we utilise mutable states in OCaml and implement the path condition as a mutable reference to a `Solver.t`. Second, we opt for *depth-first* exploration of branches, allowing us to leverage the incremental reasoning capabilities of Soteria presented above and to optimise memory sharing between branches. We do this by implementing the non-determinism monad using idiomatic OCaml iterators, such that a symbolic computation is simply an iterator over the leaves of the SE tree. Specifically, a symbolic computation is a function that takes a continuation `f : 'a → unit` and applies it to all results of type `'a` produced by the computation.

Implementing `branch_on`. With our design choices above, implementing `branch_on` is straightforward. This primitive receives a guard, two symbolic computations for the 'then' and 'else' branches, and a continuation `f`. First, the guard is simplified using the solver's contextual simplification procedure; if simplified to `true` or `false`, we immediately execute the corresponding branch. Otherwise, we (1) save the current solver state; (2) add the guard to the path condition; (3) check the satisfiability of the path condition, and if satisfiable, we pass the continuation `f` to the 'then' branch for execution; (4) backtrack the solver state to the saved state; (5) add the negation of the guard to

the path condition, and (analogously) if satisfiable, we pass `f` to the 'else' branch for execution. We present our implementation in detail in the extended version [59] for the interested reader.

3.3 The Data Module

We give a high-level overview of `Soteria.Data` as an example of a module that provides reusable components on top of the `Symex` interface. The `Data` module contains data structures that have been lifted to the symbolic world, together with operations lifted to symbolic computations.

Maps. We focus on `Map`, a key-value map data structure which exists in the OCaml standard library, but could not be used soundly with symbolic keys out of the box and requires a custom symbolic implementation. For instance, consider the `find_opt` operation that retrieves the value associated with a key in a map, and returns `None` if the key is absent.

```
(* OCaml standard library *)
val find_opt : 'a Stdlib.Map.t → Key.t → 'a option

(* Soteria.Data *)
val find_opt : 'a Soteria.Map.t → Key.t → 'a option Symex.t
```

Let us explain why the `Stdlib` implementation of `find_opt` is unsound when used with symbolic keys, and how we address this issue in our `Soteria.Data.Map` implementation. When retrieving the value associated with a key, the `Stdlib` implementation finds an entry in the map whose key is *syntactically* equal to the queried key (the same OCaml value). However, a symbolic computation may add a map entry for *symbolic* key $\widehat{x}$ (where $\widehat{x}$ is a symbolic variable) and later attempt to retrieve the value of symbolic key $\widehat{y}$ (a different variable, and hence a different OCaml object). As $\widehat{x}$ and $\widehat{y}$ are different *syntactically*, `find_opt` would return `None`. However, this is unsound under path condition $\widehat{x} = \widehat{y}$: the two symbolic keys are equal, and thus the entry for $\widehat{x}$ should be returned when querying for $\widehat{y}$. On the other hand, `Soteria.Data.Map` uses *symbolic equality* (exposed by the `SymEq` module type) on keys and returns a symbolic computation (`Symex.t`) that queries the path condition to determine if two keys are equal. Executing this symbolic computation under a path condition that does not constrain the two key variable returns *two branches*: one where the keys are equal and the entry is returned, and one where they are not and `None` is returned; in each branch, the path condition is updated with the corresponding constraint (`x = y` or `x != y`).

Branching and Optimisation. In theory, this symbolic implementation of `find_opt` (on the right) may lead to an exponential blowup in the number of branches when querying for a key in a map with many entries, as each entry may be equal to the queried key or not. In practice, however, that is not the case, thanks to (1) the optimisations described earlier in this section, (2) constraints that are already in the path conditions when accessing such maps, and (3) additional optimisations we have implemented in the `Data.Map` module. In particular, when querying for a key, we first check if the key is syntactically equal to any of the keys in the map, which is a cheap check that may avoid branching altogether.

```
let find_opt_sym map key =
  let rec find_bindings = function
    | [] → Symex.return None
    | (k, v) :: tl →
        if%sat Key.eq key k
        then Symex.return (Some v)
        else find_bindings tl
  in
  (* Syntactic lookup *)
  match M.find_opt key st with
  | Some v → Symex.return (Some v)
  | None →
    find_bindings (M.to_list map)
```

In general, this implementation illustrates how a call to `if%sat` does not necessarily lead to branching. In fact, in the SOTERIA$^{\text{RUST}}$ example we describe later in §7 (where values are inserted into a `BTreeSet`), we record more than 300M calls to `if%sat`, of which only 4683 branches are feasible.

Other Data Structures. In theory, any data structure from the OCaml standard library can be adapted in this way and used soundly in Soteria, so long as any of its operations that manipulate symbolic values are lifted to sound symbolic computations using the `Symex` interface. For instance, we also provide a `Range` module to represent pairs of symbolic integers and reason about their ordering and inclusions.

Parametricity. `Soteria.Data` also provides a series of module types (OCaml's analogue of interfaces/typeclasses) for symbolic abstractions. For instance, it provides an interface for users to describe "an integer" or "values that can be compared for equality" (used in the `Map` implementation to characterise the type of keys). This is particularly useful since Soteria leaves the choice of the representation of symbolic values to the user, and thus the `Data` module cannot assume e.g. a specific representation of symbolic integers. Note that the data structures in `Data` are parametrised by the choice of an SE monad `Symex` (itself parametrised by a solver and symbolic values). As such, users may swap the choice of values, solvers, or even the order of branch exploration, *without* having to reimplement these data structures.

4 Soteria^{Rust}: Soteria for Rust

We instantiate Soteria to develop Soteria^{Rust}, a *symbolic execution (SE) engine for Rust.* As we show below, the performance of Soteria^{Rust}, both in terms of speed and the bugs detected, is *comparable or often better* than the state of the art tool Kani [51] and passes a very large fragment of the tests for Miri [12], the reference (concrete) interpreter for Rust. Soteria^{Rust} targets more Rust features than Kani: unlike Kani, it accounts for *Tree Borrows* [58], the aliasing model of Rust. Unlike Miri, Soteria^{Rust} performs *symbolic* rather than *concrete* testing. To our knowledge, Soteria^{Rust} is the *first* SE engine that supports Tree Borrows. We proceed with a description of the Soteria^{Rust} engine (§4.1) followed by an in-depth evaluation of its performance (§4.2).

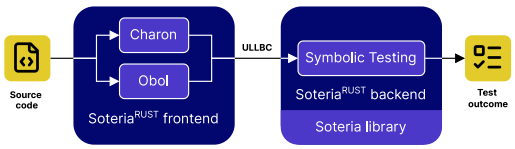


Fig. 2. The Soteria^{Rust} architecture

4.1 The Soteria^{Rust} Engine

Soteria^{Rust} operates on ULLBC (Unstructured Low-Level Borrow Calculus) code [26], a representation of MIR (Mid-level Intermediate Representation) designed with formal analysis in mind.

As shown in Fig. 2, Soteria^{Rust} has two frontends for parsing Rust: (1) *Obol*, our own front-end which directly hooks into the Rust compiler public API to extract ULLBC, and is tailored for efficiently extracting monomorphised code; and (2) Charon [25], which is currently slower and supports a smaller surface of the Rust language, but supports polymorphic code and is therefore suitable for our planned future work on enabling polymorphic symbolic execution in Soteria^{Rust}. We also provide a Kani-compatible API, allowing users to run Soteria^{Rust} as a drop-in replacement for Kani [51] and run (existing) Kani tests with Soteria^{Rust}.

```
let exec_stmt stmt =
  match stmt.kind with
  | Nop → ok ()
  | Assign (place, rval) →
    let* ptr= resolve_place place in
    let* v = eval_rvalue rval in
    State.store ptr place.ty v
```

We design symbolic values by lifting `Bv_values` to Rust as defined in the extended version [59]. The Soteria^{Rust} interpreter straightforwardly traverses the ULLBC AST from the found entry points (a `main` function, or functions annotated with `#[kani::proof]`), executing statements and evaluating expressions. The execution is wrapped inside a state monad, itself wrapped in the `Symex`

monad of Soteria, allowing the state and variable store to be threaded through SE. The lifting above shows an excerpt of the interpreter for executing the `Nop` and `Assign` statements.

The Soteria^{Rust} State. The Soteria^{Rust} interpreter is parametric on a module that implements the Rust `State` interface, which defines the type of the state and pointers, as well as symbolic computations for interacting with the state using these pointers.

Currently, Soteria^{Rust} provides one state implementation. Its pointers, defined across, are structures comprising a `BV_Value` pointer (a pair of symbolic location and offset, in the style of CompCert [34]), a Tree Borrow tag (all Tree Borrow-related terms are explained below), and symbolic values corresponding to the alignment and size of the allocation the pointer points to. Keeping track of the alignment and size within the pointer itself enables us to efficiently check for various kinds of pointer arithmetic mistakes, without repeatedly retrieving that information from the state. States (simplified for presentation) are symbolic maps (implemented using `Soteria.Data.Map`) from symbolic locations to memory objects. Each memory object is a pair comprising a *tree block* and a Tree Borrow. A tree block (unrelated to Tree Borrow) is a data structure for efficient and automated reasoning about symbolic byte arrays, following the design of Ayoun [5], and extended to annotate each byte with a Tree Borrow state. The implementation of this state was greatly simplified by the data structures provided by Soteria and the ability to define our own OCaml data structures and types such as Tree Borrows, Tree Borrow blocks and Tree Borrow tags.

```
type ptr = {
  ptr   : BV_Value.(ptr t);
  tag   : Tree_borrow.tag;
  align : BV_Value.(nonzero t);
  size  : BV_value.(int t);
}
type state = (* simplified *)
(loc, Tree_block.t * Tree_bor.t)
  Soteria.Data.Map.t
```

Tree Borrows. With these data structures in place, implementing Tree Borrows is surprisingly simple. In the Tree Borrow model [58], reading from or writing to a byte with Tree Borrow state s through a pointer with tag t requires taking a transition in a state machine depending on s, t, and the structure of the tree T of the Tree Borrow associated with the allocation. We can do this in OCaml with a single a pattern match over the state s and the kind of transition (determined by t and T) to determine the next state s', when the operation is allowed; otherwise (when it is not allowed), we raise an error. By contrast, had we compiled Rust to an IL, we would have had to encode this entire logic in the IL as well, significantly increasing the complexity of its implementation.

Consider the example across by Villani et al. [58]: two mutable references `x` and `y` are created on line 5, aliasing the same address. This is forbidden in safe Rust, but bypassed here using raw pointers and the `unsafe` keyword. When `x` is written to on line 6, Tree Borrows dictate that the state of each modified byte is updated to *disable* access through all other tags, including that of `y`. As such, using `y` on line 7 triggers UB, which Soteria^{Rust} correctly detects by checking the state of the tag associated with `y`.

```
1 fn main() {
2   let mut root = 42;
3   let ptr = &mut root as *mut i32;
4   let (x, y) = unsafe {
5     (&mut *ptr, &mut *ptr) };
6   *x = 13;
7   *y = 20; // UB: y is disabled
8 }
```

Intrinsics. When designing SE engines, one common challenge is the need to support numerous functions that are built into the language in addition to all language features. Rust is no exception and defines, at the time of writing [45], a total of 223 *intrinsics*. These functions

```
#[rustc_intrinsic]
pub const unsafe fn copy_nonoverlapping<T>(
    src: *const T, dst: *mut T, count: usize)
```

```
val copy_nonoverlapping :
  t:ty → src:ptr → dst:ptr →
  count:BV_value.(int t) → unit ret
```

allow for all kinds of low-level operations such as floating point arithmetic or atomic manipulation. Unfortunately, these intrinsics are rather unstable, which could constitute a maintenance burden over time without careful design.

To address this challenge in Soteria$^{\text{Rust}}$, we use the Rust signatures of these intrinsics to automatically generate the corresponding OCaml signatures, together with all the necessary boilerplate to integrate them into the interpreter. We then lift these OCaml signatures to SE computations, receiving symbolic arguments and returning symbolic computations (hidden behind the `ret` type). This design ensures that the signatures are always in sync with their ever-evolving Rust definitions, and streamlines the addition of each new intrinsic.

Note that the developer still needs to implement the actual intrinsic itself, matching the generated interface. In most cases, the implementation is straightforward: the average implementation is 5.6 lines of OCaml code across 165 implemented intrinsics, with the longest implementation being 35 lines of code. The challenge of handling intrinsics thus lies in keeping up to date with their ever-evolving signatures (which our design addresses) rather than in their implementation.

4.2 Evaluation

We evaluate Soteria$^{\text{Rust}}$ against Kani and Miri. Miri is a concrete interpreter for Rust and can detect undefined behaviours (UBs). It is the *de facto* tool for testing Rust code for UBs and is used extensively by the Rust compiler team. Both Kani and Miri come with their own test suites, and we thus evaluate all three tools on both Kani and Miri suites (§4.2.1). We further compare all three tools on a fresh test suite that is not biased towards either tool (§4.2.2). Finally, we compare Soteria$^{\text{Rust}}$ and Kani on the tests of a real-world Rust library, `finetime` [56] (§4.2.3). We run all tests on a MacBook Pro (M4 Pro, 14 cores, 32GB RAM).

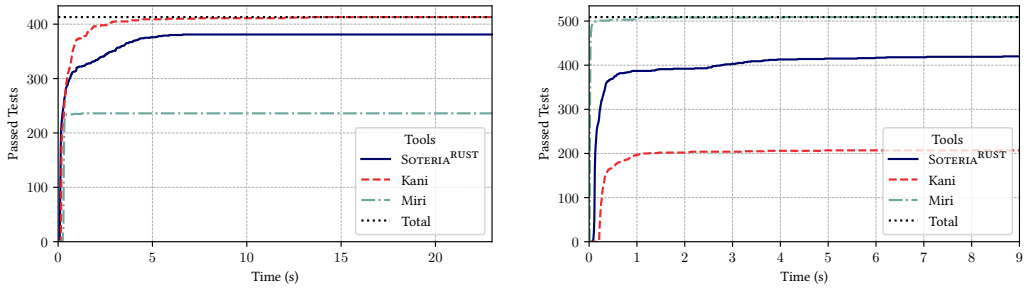
4.2.1 The Kani and Miri Test Suites. The two suites comprise 922 tests in total: 413 Kani (commit `6c2c4f0`) and 509 Miri (commit `ec775c1`) tests. For the Kani suite, we run Kani as intended (with `//kani-flags` annotations in the tests); we run Miri by replacing `#[kani::proof]` annotations with `#[test]` and assuming they are concrete (if any Kani built-in is reached, Miri gives up, as it does not support *symbolic* execution). We run Soteria$^{\text{Rust}}$ with (1) unlimited fuel, ensuring *all* feasible paths are explored, (2) Kani compatibility enabled, linking against the Kani library wrapper, and (3) memory leak and aliasing checks disabled (as Kani does not check for them).

For the Miri suite, we run Miri as intended, and we run Soteria$^{\text{Rust}}$ as is, since Miri compatibility is built-in. We run Kani with flags `uninit-checks` and `valid-value-checks` to enable uninitialised memory access and validity checks, ensuring a similar level of thoroughness to Miri and Soteria$^{\text{Rust}}$.

Kani (resp. Miri) takes 18s (resp. 4s) to run its longest test, so we set a timeout of 23s (resp. 9s) for all three tools on the Kani (resp. Miri) suite. We classify the test outcomes into four categories: *pass* (where the test outcome matches the expected outcome), *fail* (when the test outcome disagrees with the expected outcome), *unsupported* (if the tool crashes or raises an error, or a feature is not supported), and *timeout*.

We present the results for each test suite in Fig. 3 (above), as well as the number of tests passed by each tool over time (survival graph) in Figs. 3a and 3b. Unsurprisingly, both Kani and Miri pass all tests in their respective suites; for both suites, Soteria$^{\text{Rust}}$ performs the second best and far outperforms Kani (resp. Miri) on the Miri (resp. Kani) suite. Moreover, Soteria$^{\text{Rust}}$ passes the highest number of tests across both suites (86.9%). Soteria$^{\text{Rust}}$ does not support 23 tests in total across both suites, due to gaps in the frontend (Obol does not support `TypeId`), and to unsupported features in the engine (`&dyn Trait` method calls where the first argument is `Self`, as it is unsized). Additionally, Soteria$^{\text{Rust}}$ fails 79 tests, i.e. it either detects an error where there is none, or misses an error. These failures, which predominantly happen in the Miri suite, are due to UB behaviours

Tool	Kani Suite				Miri Suite				Total
	#Pass	#Fail	#Unsup.	#Timeout	#Pass	#Fail	#Unsup.	#Timeout	% Pass
Kani	413	0	0	0	207	106	190	6	**67.2%**
Miri	236	8	169	0	509	0	0	0	**80.8%**
SOTERIA$^{\text{RUST}}$	381	18	9	5	420	61	14	14	**86.9%**



(a) Kani suite (b) Miri suite

Fig. 3. The Kani and Miri test suite results (above); number of tests passed over time by each tool (below)

```rust
let x: u128 = kani::any();
let mut count: u32 = 0;
for i in 0..u128::BITS {
  let bit = x & (1 << i);
  if bit != 0 { count += 1; }
}

assert!(count == ctpop(x));
```

(a) Original test from the Kani suite

```rust
let x: u128 = kani::any();
let mut count: u32 = 0;
for i in 0..u128::BITS {
  let bit = x & (1 << i);
  count = count.wrapping_add(
    (bit != 0) as u32);
}
assert!(count == ctpop(x));
```

(b) Modified test, verifiable by SOTERIA$^{\text{RUST}}$

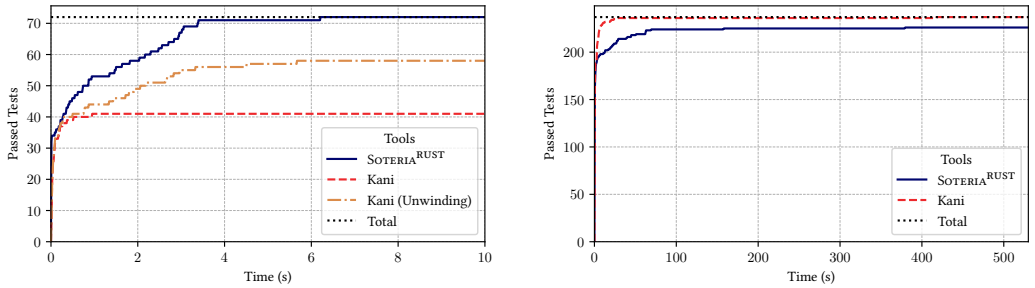
Fig. 4. Example of a biased test in the Kani test suite, and a modification to make it verifiable by SOTERIA$^{\text{RUST}}$

not being checked, e.g. not checking for address overlap between function arguments and return address, or not checking the validity of `&dyn Trait` upcasting. The failures in Kani are mostly due to trigonometric intrinsics (such as `cosf32` or `log10f32`) being implemented with insufficient precision, or to limitations around float operations (SMT-LIB does not handle NaN patterns accurately).

Thanks to SOTERIA, a *first-year PhD student* developed a competitive SE engine for Rust in merely eight person-months. The limitations of Miri are due to its concrete nature and are hardly fixable, and those of Kani are due to its incomplete support for Rust semantics, which though fixable, require a significant engineering effort. By contrast the limitations of SOTERIA$^{\text{RUST}}$ are due to missing features that can be added with little friction, given the modular design of SOTERIA.

Test Bias. We note that both Kani and Miri suites are biased towards their respective tools. For instance, consider the test for the `ctpop` intrinsic in the Kani suite, shown in Fig. 4a, which counts the number of 1 bits in a given number `x`. Both Kani and SOTERIA$^{\text{RUST}}$ pass this test, but the test is biased towards the strengths of Kani. Specifically, despite the branch (if-condition) in the loop, CBMC easily merges the branches into a single path. By contrast, SOTERIA$^{\text{RUST}}$ currently has no such branch merging capacity and branches on every loop iteration, leading to 2^{128} paths for a 128-bit integer. However, as shown in Fig. 4b, a simple modification (where we drop the 'if' and use `wrapping_add` to avoid checking for overflow) eliminates branching and allows SOTERIA$^{\text{RUST}}$ to

Tool	BinaryHeap	BTreeMap	LinkedList	Option	Result	Vec	VecDeque	Total
Kani	1	1	0	14	16	8	1	**41**
Kani (unwind.)	4	2	7	15	16	9	5	**58**
SoteriaRust	6	8	8	15	16	11	8	**72**
Total Tests	**6**	**8**	**8**	**15**	**16**	**11**	**8**	**72**



(a) Tests passed in the DS suite over time

(b) Tests passed in the `finetime` crate over time

Fig. 5. The DS suite results with a timeout of 10s (above); tests passed over time by each tool (below).

verify the property in significantly less time. SoteriaRust runs the modified test faster than Kani: Kani passes the first test in 24.5s and the modified test in 14.7s, while SoteriaRust times out on the first test and passes the modified test in 1.1s.

Similarly, as we show in §4.2.2, code patterns such as heap-intensive code and pointer manipulation (which are ubiquitous in real-world Rust code) degrade the performance of Kani significantly. Our case study in §4.2.2 suggests that SoteriaRust handles these patterns more efficiently. We discuss path-merging versus path-exploration more extensively in §7.

4.2.2 Data Structures Tests (DS Suite). To showcase the performance of SoteriaRust on real, unbiased code, we wrote a number of symbolic tests for several Rust data structures, including `Vec`, `LinkedList`, `BTreeMap`, `BinaryHeap`, `VecDeque`, `Option`, and `Result`. The tests (72 in total) create data structures with an arbitrary size up to a bound and perform a number of operations such as inserting, removing, and accessing symbolic elements. We present their results in Fig. 5, run with a 10-second timeout, and their survival graph in Fig. 5a. As these tests are inherently symbolic, we did not run Miri on them (Miri cannot run symbolic tests).

While SoteriaRust passes *all* tests in 6.6 seconds, Kani only passes 41 (resp. 58) without (resp. with) loop unwinding annotations (`#[kani::unwind(N)]`); SoteriaRust ignores these annotations. We use the minimum unwinding value that ensures no branches are missed. Furthermore, we run Kani without validity and undefined memory access checks; with these enabled, the tool gives up on the majority of tests, due to incomplete support of uninitialised memory checks. We run Kani using CaDicaL [8] as the underlying SAT solver; we observed no notable difference when using Kissat [9] instead. As shown, SoteriaRust is faster on DS tests, while performing a deeper analysis with validity, undefined memory access, *and* Tree Borrow checks.

We highlight that these tests were written *with no knowledge of Kani limitations* and *with no special accommodations to advantage SoteriaRust*. In particular, to eliminate bias towards SoteriaRust, we removed ourselves from the test writing process and used an AI pair programmer to generate them; we present the AI prompt we used in the extended version [59]. The only adjustment we made was to the size bounds of the data structures to ensure the tests complete in a reasonable time.

4.2.3 Real World Usage. We compare Soteria$^{\text{Rust}}$ with Kani on `finetime` [56], an efficient and high-precision time keeping library. We used the version from commit `f5962b8`. The crate has 237 symbolic tests using the Kani interface, which test the panic-freedom of various calendar conversions, the accuracy of rounding durations, and the correctness of round-trip conversions between time representations. We note that the `finetime` tests were written for Kani, and they may thus be biased towards its strengths.

We present the results in Fig. 5b. Neither Soteria$^{\text{Rust}}$ nor Kani fail; Kani passes the slowest test in 515s, while Soteria$^{\text{Rust}}$ times out on 11 tests (with timeout set to 530s). In all 11 cases, Soteria$^{\text{Rust}}$ gets stuck waiting for the solver (Z3 [18]) to resolve queries involving signed remainder and division operations on bitvectors, which are a known limitation of SMT solvers.

Despite limitations around signed operations, Soteria$^{\text{Rust}}$ is competitive with Kani on real-world code, even when bit operations are heavily used. We plan to improve the support for these operations in future work, with formally verified reductions to simpler operations [7].

We have also used an AI agent to write symbolic tests for a variety of real-world Rust libraries. This allowed us to find a case of potential UB in the `hashbrown` crate [48], which implements hash maps and sets, and is used in the Rust standard library; it is also the second most downloaded Rust crate [49]. We submitted a pull request to fix the issue, which has since been merged.[1]

5 Soteria$^{\text{C}}$: Soteria for C

We develop Soteria$^{\text{C}}$, a symbolic execution (SE) engine for C that can perform *both* whole program symbolic testing (WPST) and fully automated, bi-abductive bug detection. We present an overview of its infrastructure and evaluate its performance and bug-finding capabilities against existing tools.

5.1 Architecture and Engine

We use Cerberus [39] to parse C and de-sugar it into AIL, which is close to the abstract C source, but with all types checked and canonicalised. Much like Soteria$^{\text{Rust}}$, Soteria$^{\text{C}}$ is defined by recursively traversing the source AST. The two core functions of the engine, `eval_stmt` (evaluating a C statement) and `eval_expr` (evaluating an expression) return an `InterM.t` monad value, which is obtained by applying the state monad combinator to the `Symex.Result` monad. Intuitively, `InterM.t` is an SE monad where each branch may either be successful or erroneous, and where each branch carries an additional state corresponding to the C heap. For example, evaluating `if` is as shown below.

First, the guard `cond` is evaluated, obtaining the symbolic value `v`. C expressions evaluate to an 'aggregate' value, which is either a base value (integer, float or pointer) or a structure; therefore, `v` must be cast to a symbolic boolean using `cast_aggregate_to_bool`. Finally, we call `if%sat` to evaluate the 'then' and 'else' statements depending on the satisfiability of `v`.

```
let rec exec_stmt = function
| AilSif (cond, then_stmt, else_stmt) →
  let* v = eval_expr cond in
  let* v = cast_aggregate_to_bool v in
  if%sat v then exec_stmt then_stmt
  else exec_stmt else_stmt
| ...
```

Bi-abduction and Bug Finding. We implement 'fix-from-error' bi-abduction, adapting existing work [5, 23, 35] to Soteria. The main novelty of our approach lies in formulating the transformation from standard SE to bi-abductive SE as a state monad transformer, which we distribute as part of the Soteria library in the `State` module. We elide the details of this transformer for lack of space, though it is defined and proven in Ayoun [5].

[1] https://github.com/rust-lang/hashbrown/pull/692

Table 1. Whole-program symbolic testing (WPST) results on Collections-C with shortest times **highlighted**.

Folder	Tests	CBMC (s)	Gillian-C (s)	SOTERIA^C (s)
array	21	131.45	15.43	**11.39**
deque	34	278.47	23.89	**11.41**
list	37	134.94	25.85	**23.73**
pqueue	2	2.27	1.42	**1.12**
queue	4	4.33	2.73	**1.42**
rbuf	3	3.13	2.08	**1.00**
slist	37	48.03	25.40	**15.28**
stack	2	2.16	1.36	**0.77**
treeset	6	180.00	4.28	**2.85**
treetable	13	390.00	8.97	**5.57**
Total	159	1174.78	111.41	**74.54**

As with Pulse [32], automatic bug finding works by first running bi-abductive SE to generate function summaries, and then applying a 'manifest bug' criterion to the summaries that have an erroneous post-condition to determine whether a bug should be reported to the user.

5.2 Evaluation

We evaluate SOTERIA^C on Collections-C, used by Gillian-C to evaluate WPST and bi-abduction [5, 22, 35]. We run all benchmarks on a MacBook Pro (M4 Pro, 14 cores, 32GB RAM).

Whole-Program Symbolic Testing (WPST). We run SOTERIA^C, Gillian-C and CBMC on the same 159 symbolic tests as Fragoso Santos et al. [22], with a timeout of 30s per test, and compare their results. We run the three tools with options that, we believe, provide a fair comparison with SOTERIA^C in terms of number of checks.[2] In particular, all three tools are run such that there is no limit on the number of explored paths, meaning that *all* feasible branches are explored (the tests are written to be bounded). We present the performance results in Table 1.

Table 2. The WPST result categories on Collections-C

Tool	Pass	Fail (FN)	Fail (FP)	Crash	Timeout
SOTERIA^C	**159**	**0**	**0**	**0**	**0**
Gillian-C	156	3	**0**	**0**	**0**
CBMC	124	2	2	1	30

Note that these tests were written by the Gillian-C authors, and thus they are subject to *bias* (towards Gillian-C), as we described above in §4. SOTERIA^C runs *faster* than Gillian-C on *all tests*; this is despite the fact that, unlike SOTERIA^C, Gillian-C does not perform integer overflow checks. Both SOTERIA^C and Gillian-C run *significantly faster* than CBMC. However, as these tests are (potentially) biased towards Gillian-C, further evaluation is needed to compare the tools on tests written for SOTERIA and for CBMC. Moreover, CBMC times out on 30 of the (159) tests (with the timeout of 30s). As such, 900s of the total ~1190s taken by CBMC is spent on timed-out tests.

Moreover, as shown in Table 2, SOTERIA^C performs a *more accurate* analysis than both Gillian-C and CBMC. Specifically, all bugs found by Gillian-C and CBMC were also found by SOTERIA^C; that is, to our knowledge SOTERIA^C has no false negatives (FNs, missed bugs which were found by at least one other tool). On the other hand, Gillian-C and CBMC have three and two FNs, respectively. In the case of Gillian-C, two of the FNs occur because the Gillian compiler *optimises away* bugs

[2]Specifically, we run Gillian-C with default options, and we run SOTERIA^C with `--alloc-cannot-fail --infinite-fuel --havoc-undef --cbmc`. We run CBMC 6.8.0 (Nov 5th 2025) with the `--bounds-check --pointer-check --div-by-zero-check --pointer-primitive-check --unwind 10 --drop-unused-functions --signed-overflow-check --pointer-overflow-check --float-overflow-check --unwinding-assertions --no-malloc-may-fail`.

due to accessing uninitialised memory; the reason for the third FN in Gillian-C is unclear to us. Similarly, CBMC misses the same two bugs as Gillian-C due to accessing uninitialised memory; this is because CBMC does not check for accesses to uninitialised memory.

Moreover, Soteria^{C} and Gillian-C have no *false positives* (FPs), do not crash and do not time out. However, as discussed above, CBMC times out on 30 of the tests and crashes on one. Lastly, CBMC reports two FPs; i.e. CBMC reports two bugs where there are none. This is because CBMC performs over-approximate (OX) analysis that is not always precise enough for bug detection.

Automatic Bug Finding. We evaluate the performance and ability of Soteria^{C} to find bugs in Collections-C automatically, and compare it with Pulse [32] (version 1.2.0). We run Pulse with `--pulse-only -j1` to activate only the Pulse analysis with a single thread. Both Soteria^{C} and Pulse find the same bug, which we have confirmed to be a *true positive*; we submitted a pull request to fix it upstream, which has been accepted. Soteria^{C} is competitive with Pulse in terms of performance.

Thanks to the `Stats` utility of Soteria, we can provide a detailed break-down of the analysis performed by Soteria^{C} in 2.57s. Specifically, Soteria^{C} spends: 0.35s parsing and desugaring the source code; 1.35s analysing the *578 functions* in the codebase (containing assorted control flow) and generating *2364 summaries* (each summary corresponds to one explored satisfiable path); and 0.60s performing "manifest bug" analysis on these summaries, isolating the single true positive bug report. The whole process runs through 54,925 `if%sat` calls, performing 10,503 sat checks that are not immediately simplified.

Table 3. Results of bi-abductive bug finding on Collections-C.

Tool	Time (s)	Bugs found
Soteria^{C}	2.57	1
Pulse	2.52	1

We do not compare Soteria with Gillian-C because Gillian-C does not implement a 'manifest bug' analysis (needed for identifying true positive bug reports) and only produces function summaries. Specifically,Gillian-C produces (on Collections-C) over 4000 bug summaries, and thus (without a manifest bug analysis) manual inspection is infeasible. Moreover, Gillian-C runs significantly slower: Gillian-C takes ~62.8s, while Soteria^{C} and Pulse take ~2.57s and ~2.52s, respectively. The data provided in this paragraph for Gillian-C is extracted from Lööw et al. [35].

In the future we would like to evaluate Soteria^{C} on a larger and more diverse set of C benchmarks. Nevertheless, in merely three person-months, we implemented a symbolic C engine that can perform WPST and bi-abductive bug finding. This is thanks to the flexibility and reusability of Soteria.

6 Formalisation

We formalise the symbolic execution (SE) monad that underpins Soteria. Our formalisation is closely inspired by existing formalisations [28], with the main novelty being the support for under-approximating monadic SE. It provides a formal, mental model of the Soteria implementation, and client implementations of the `Symex` interface must be correct against this formalisation.

Values and Symbolic Variables. Recall from §2 that SE uses *symbolic variables* to represent multiple values at once. Soteria is parametric on the values desired for execution, supplied by the user. As such, we assume the user provides a set of values, Val, together with a set of *sorts* $\mathbb{T} \ni \tau$ with $\mathbb{T} \triangleq \mathcal{P}(\mathit{Val})$. We also assume a countably infinite set of variable identifiers, VarId, and define symbolic variables $\widehat{x}, \widehat{y} \in \widehat{\mathcal{X}} \triangleq \mathit{VarId} \times \mathbb{T}$ (as a convention, we denote symbolic entities with a hat) as pairs of a variable identifier and a sort. We write $\widehat{x} : \tau$ to denote that $\widehat{x}$ is of sort τ.

Symbolic variables and concrete values are connected through *symbolic interpretations*, $\varepsilon \in \mathcal{I} \triangleq \widehat{\mathcal{X}} \rightharpoonup \mathit{Val}$, defined as partial maps from symbolic variables to values, enforcing the following *well-sortedness invariant*: for all $\widehat{x}, v$, if $\widehat{x} : \tau$ and $\varepsilon(\widehat{x}) = v$, then $v \in \tau$.

Symbolic Abstractions. Given a set A, a *symbolic abstraction* over A, $\widehat{a} \in \widehat{A}$, is an object that is interpreted as a set of elements of A under some ε. Formally, there must exist a relation $\models \subseteq \mathcal{I} \times A \times \widehat{A}$ such that $\varepsilon, a \models \widehat{a}$ means that a is one of the possible values represented by $\widehat{a}$ under interpretation ε.

A simple example of symbolic abstraction is *symbolic values*, which are symbolic abstractions over the set of values *Val*. Symbolic values can also be built by lifting operators over values to symbolic values in the natural way. For instance, the value $\widehat{x}+1$ is the symbolic value that represents the singleton set $\{\varepsilon(\widehat{x}) + 1\}$ under interpretation ε. Similarly, symbolic maps as presented in §3.3 are symbolic abstractions of finite partial maps. In general, the `Data` module of Soteria contains implementations for symbolic abstractions over data stuctures.

Symbolic Booleans and Solvers. To define the SE monad we need the notion of *symbolic booleans*, $\widehat{\mathbb{B}} \ni \pi$, which are symbolic abstractions over booleans $\mathbb{B}$. A symbolic boolean π is *satisfiable*, denoted $\mathsf{SAT}(\pi)$, if there exists an interpretation ε such that $\varepsilon, \mathsf{true} \models \pi$. Conversely, it is *unsatisfiable*, denoted $\mathsf{UNSAT}(\pi)$, if there exist no such interpretation.

A *solver* is an oracle that can determine the satisfiability (or unsatisfiability) of symbolic booleans. In practice, however, solvers are not perfect. For instance, SMT solvers are often based on incomplete theories and may return UNKNOWN when queried. To address this, we interpret UNKNOWN results *approximately*, based on the analysis *mode* m, which may be under- (UX) or over-approximate (OX). Specifically, SE in UX must not deem an infeasible path as feasible, and thus it must interpret UNKNOWN as UNSAT. Conversely, SE in OX must interpret UNKNOWN as SAT. We generalise this by defining *m-approximate* solvers SAT_m, where $m \in \{\mathsf{OX}, \mathsf{UX}\}$, such that:

$$\mathsf{SAT}(\pi) \Rightarrow \mathsf{SAT}_{\mathsf{OX}}(\pi) \qquad \text{and} \qquad \mathsf{SAT}_{\mathsf{UX}}(\pi) \Rightarrow \mathsf{SAT}(\pi)$$

Remark 1. Another practical consideration that tends to be ignored by existing formalisations of symbolic execution is that of *partial functions*. Specifically, SMT-LIB-compatible solvers such as Z3 or CVC5 consider partial functions as *unspecified* when applied outside their domain. For instance, Z3 deems the expression $\widehat{y} = \widehat{x}/0$ to be *satisfiable* for all $\widehat{x}$ and $\widehat{y}$, because division by zero makes the expression unspecified, not undefined. Therefore, the symbolic boolean $\widehat{x}/0$ must be viewed as a symbolic abstraction that can be interpreted to *any integer* under any interpretation.

Branches. A *branch*, $\langle a \mid \pi \rangle \in \langle A \mid \widehat{\mathbb{B}} \rangle$, is a pair comprising a result value $a \in A$ and a *path condition* (a symbolic boolean) $\pi \in \widehat{\mathbb{B}}$. We can now define the SE monad as follows:

$$\mathsf{Symex}(A) \triangleq \widehat{\mathbb{B}} \to \mathcal{P}(\langle \widehat{A} \mid \widehat{\mathbb{B}} \rangle) \qquad \mathtt{return}(a) \triangleq \lambda\pi.\ \{\langle a \mid \pi \rangle\}$$
$$\mathtt{bind}(m, f) \triangleq \lambda\pi.\ \{\langle b \mid \pi'' \rangle \mid \langle a \mid \pi' \rangle \in m(\pi) \wedge \langle b \mid \pi'' \rangle \in f(a, \pi')\}$$

The `return` function returns a single branch containing the given value and does not modify the path condition. The `bind` function takes each branch of the first computation and feeds its value and path condition to the second computation, collecting all resulting branches. We can similarly define other operations in the `Symex` interface; e.g. we can define `nondet` and `if%sat` as follows:

$$\mathtt{nondet}(\tau) \triangleq \lambda\pi.\ \{\langle \widehat{x} \mid \pi \rangle\} \ \mathtt{where}\ \widehat{x} : \tau \text{ is fresh}$$

$$\mathtt{branch\_on}(\widehat{b}, t, e) \triangleq \lambda\pi.\ \left(t(\pi \wedge \widehat{b}) \text{ if } \mathsf{SAT}_m(\pi \wedge \widehat{b}) \text{ else } \emptyset\right) \cup \left(e(\pi \wedge \neg\widehat{b}) \text{ if } \mathsf{SAT}_m(\pi \wedge \neg\widehat{b}) \text{ else } \emptyset\right)$$

That is, as described in §3, symbolically executing a branch adds the guard $(\widehat{b})$ and its negation $(\neg\widehat{b})$ to the path condition and executes the corresponding branch if the resulting path condition is satisfiable; otherwise, it discards the branch. If both are satisfiable, it explores both branches.

Soundness. We define soundness for a *SE computation*, i.e. a function $\widehat{A} \to \mathsf{Symex}(\widehat{B})$, with respect to a nondeterministic computation, i.e. a function $A \to \mathcal{P}(B)$. We define both OX and UX soundness (respectively desirable for verification and bug-finding) in the extended version [59]. Let $f : A \to \mathcal{P}(B)$ be a nondeterministic function, $\widehat{A}$ and $\widehat{B}$ be symbolic abstrations over A and B,

and $\widehat{f} : \widehat{A} \to \text{Symex}(\widehat{B})$. Intuitively, a function $\widehat{f}$ is OX-*sound* with respect to f, written $\widehat{f} \preceq_{\text{OX}} f$, if each outcome of f is covered by a branch of $\widehat{f}$ (is also an outcome of $\widehat{f}$). We define UX-soundness analogously; we elide the formal definitions here and present them in the extended version [59].

A simple example of a both OX- and UX-sound symbolic computation is `nondet`, which is sound with respect to the nondeterministic function that returns all values of the given sort. More complex symbolic computations are built by composing simple symbolic computations using `bind` and `branch_on`. We show that these two operations *preserve* soundness. Specifically, Theorem 6.1 (proof in the extended version [59]) states that (1) composing two m-sound symbolic computations using `bind` yields an m-sound symbolic computation (this ensures the soundness of sequentially composing computations); and (2) `branch_on` is a symbolic lifting of the concrete `if...else...` construct.

Theorem 6.1 (Soundness Preservation). *For all $f, \widehat{f}, g, \widehat{g}, a, \widehat{a}, b, \widehat{b}$ and all modes $m \in \{\text{OX}, \text{UX}\}$:*

(1) if $\widehat{f} \preceq_m f$ and $\widehat{g} \preceq_m g$, then $\widehat{g} \gg\!\!= \widehat{f} \preceq_m g \gg\!\!= f$, where $p \gg\!\!= q \triangleq \lambda x.\ \texttt{bind}\ p(x)\ q$, denoting the Kleisli composition ('fish') operator.

(2) Let $h \triangleq \texttt{if}\ b\ \texttt{then}\ f\ a\ \texttt{else}\ g\ a$ and $\widehat{h}\ (\widehat{b}, \widehat{a}) \triangleq \texttt{branch\_on}\ \widehat{b}\ (\widehat{f}\ \widehat{a})\ (\widehat{g}\ \widehat{a})$. If `branch_on` uses an m-approximate solver, $\widehat{f} \preceq_m f$ and $\widehat{g} \preceq_m g$, then $\widehat{h} \preceq_m h$.

7 Discussion, Limitations and Future Work

We begin by discussing the advantages of two key design decisions we made in Soteria, namely (1) implementing Soteria as a shallowly-embedded library in OCaml; and (2) implementing "symbolic execution" (SE), rather than "symbolic compilation" that performs path merging and generates a single formula. We then proceed with discussing the limitations of Soteria, which naturally lead to possible avenues of future work.

7.1 Key Design Decisions in Soteria

Soteria in OCaml. Unlike most reusable SE, which provide a single intermediate language (IL) to analyse many languages, Soteria is a implemented in OCaml, providing base abstractions (the SE monad) and reusable components (symbolic computations and data structures) to build custom SE engines for different languages. While we have demonstrated that this method is effective (at least for Rust and C), we point out one additional advantage of this approach.

Specifically, by forgoing an IL, we can leverage all features of our host language (OCaml) when implementing engines. For instance, in our implementation of `BV_value`, we make extensive use of OCaml's ability to attach *ghost types* to values, allowing us to ensure e.g. that ill-typed expression cannot be constructed. Doing so in an IL would require implementing a custom type system from scratch, requiring substantially more work and likely leading to a reduced expressiveness in comparison to OCaml's mature type system. Another example is our use of OCaml's module system which allows us to use functors, thereby seamlessly enables us to make an engine parametric on various choices, including the choice of the symbolic state representation, or the symbolic pointer representation. Using an IL, one would have to embed this parametricity in the IL itself.

Symbolic Execution versus Symbolic Compilation for Bug Finding. Soteria performs symbolic *execution*: it explores all feasible paths of the program separately and generates a formula for each path. An alternative approach is to perform symbolic *compilation*, where the engine performs path merging and generates a single formula for the whole program. Nevertheless, we believe it is possible in Soteria to implement path merging on a given object of type `'a Symex.t` (a computation that yields a value of type `'a`), as long as a function `merge: Value.t → 'a → 'a → 'a` is provided; this is in line with the approach taken in Grisette [37].

However, existing data suggests that while symbolic compilation shines when merging path conditions over simple values such as bitvectors (as in the `ctpop` example of Fig. 4), it does not

perform as well when merging path conditions over expressions that characterise more complex data structures, e.g. objects in the heap, which can overwhelm the solver. An example of the latter is inserting n non-deterministic integers into a Rust `BTreeSet` and checking that the resulting set is ordered. For $n = 3$, Soteria$^{\text{Rust}}$ explores 13 paths in 2.09s, while Kani merges all paths into a single SAT instance of ~50M variables, taking 691s to solve; for $n = 6$, Soteria$^{\text{Rust}}$ explores 4,683 paths in 456s, while Kani exceeds a two-hour timeout (see the extended version [59] for more details).

Moreover, we envision Soteria being used primarily to implement *bug detection* (rather than *verification*) tools, where path merging may lead to imprecise abstractions [2]. In the bug detection setting we believe that path exploration is more effective, and it is the approach taken by the industrial bug detection engine Infer.Pulse [43, 46].

7.2 Limitations and Future Work

Semantic Coverage. Soteria$^{\text{Rust}}$ and Soteria$^{\text{C}}$ do not currently support various features such as concurrency (through the async/await constructs), FFI calls, or inline assembly. In the future, we will extend the coverage of both interpreters by exploring foreign calls between C and Rust and leveraging both interpreters. We will further explore how to leverage the guarantees offered by the semantics of Rust to support bug finding in concurrent code that uses async/await constructs.

A main roadblock in extending our coverage thus far lies with the *frontends* we use. For instance, Cerberus (our Soteria$^{\text{C}}$ frontend) does not currently support several built-in atomic operations used by widely-used libraries such as `libgit2` . These operations are not part of the C11 standard; they are compiler-specific extensions provided by Clang and GCC. Lack of support for these operations limits the applicability of Soteria$^{\text{C}}$, as it prevents extracting an AST for C programs using them.

Solvers. While we provide blueprints and a clear signature for custom new values and solvers, we believe a substantial part of the required work could be *automated* through meta-programming. In the future, we will create a DSL to specify new symbolic values, their operations, and simplifications that can be extracted to interactive theorem provers to be proven sound. We will further provide reusable types with libraries of simplifications for common symbolic values such as bitvectors. Finally, we will explore adjusting the settings and default tactics of Z3 to improve performance on our generated queries, and we will explore other solvers as well.

Polymorphic and Compositional Reasoning for Rust. We will extend Soteria$^{\text{Rust}}$ to support executing polymorphic functions, leveraging its parametric design and its Charon frontend. This will allow executing symbolic tests that model execution for all type instantiations of a polymorphic function. We will then explore compositional bug finding for Rust by leveraging this polymorphic support and the existing bi-abduction support in Soteria.

Lean. Our current Soteria implementation benefits from our extensive knowledge of OCaml for efficiency and ease of maintainability. However, since Soteria is a functional library, it can be implemented in any functional language with support for monads. In the future, we will explore re-implementing Soteria in Lean. We will explore other, better-suited optimisations, leverage Lean's powerful meta-programming capabilities, and investigate the interaction between symbolic execution and interactive theorem proving in this setting.

8 Related Work

Monadic Symbolic Execution. Previous work has explored using monads to structure SE engines, in more theoretical settings. Darais et al. [17] propose a methodology to lift a definitional interpreter for a given language written in a monadic style into various abstract interpreters, including a symbolic one, but do not explore an implementation for a real programming language, or how to optimise it. Keuchel et al. [28], on the other hand, present a monadic approach mechanised in

Rocq and implement in Katamaran, a tool for proving ISA security properties. While Katamaran comes with a mechanised proof of soundness, its main aim is to improve the proof engineering experience inside Rocq. They do this by using a deep embedding of path conditions that enables manual simplifications without resorting to meta-programming, and their monadic approach is *not executable*. Soteria, in contrast, explores an executable and *reusable* approach to monadic SE.

Owi [1] is an SE tool for Wasm that uses a similar monadic structure to Soteria. It is designed for performant, parallel exploration of multiple paths – a capability we have not yet exploited in Soteria. Unlike Soteria, Owi treats Wasm as a fixed IL for analysing programs compiled from C, Rust, Zig, and so forth. As such, unlike Soteria$^{\text{Rust}}$, Owi does not support Rust-specific features such as Tree Borrows, because the resulting Wasm code lacks the necessary information.

Abstract Interpretation for Static Analysis. There are many scalable static analysers based on abstract interpretation [14, 15, 29]. In particular, MOPSA [40, 41] is a multi-language analysis platform (currently for C and Python) that (as with Soteria) aims for modularity and reusability. For instance, it uses OCaml features such as extensible variants to enable elegant reuse of transfer functions across languages. Currently, MOPSA is a whole-program (non-compositional) static analyser and targets OX analyses, whereas Soteria primarily focuses on UX bug finding.

Semantic Framework and Symbolic Lifting. Rosette [53, 54] extends Racket [52] with solver-aided programming features, automatically lifting a concrete interpreter for a language written in Racket into a symbolic one using symbolic compilation (that is, compiling the entire program to a single SMT formula). Grisette [37] later formulated this approach, as well as various optimisations, using a monadic approach, providing a functional programming library in Haskell. In principle, one could encode analyses such as ours in Rosette and Grisette, though previous experiences have shown that this approach struggles to scale to real world code [23, 50].

Similarly, the $\mathbb{K}$ framework [47] allows one to define the formal semantics of a language using rewriting logic, and it has an SE backend for performing symbolic testing. However, defining semantics in $\mathbb{K}$ entails substantial work: the KMIR project [16], commenced in 2023, aims to provide a K semantics and SE for Rust; however, it does *not* support Tree Borrows and does not seem to be ready for use – we could not find any existing symbolic tests.

Rust Analysis Engines. There are several tools for semi-automatic verification of Rust programs [3, 4, 19, 21, 24, 26, 31, 33]. While they all provide more guarantees than Soteria$^{\text{Rust}}$, they all require *manual user annotations* to specify pre- and post-conditions, loop invariants, and at times even tactics to guide the proof. Moreover, *none* support reasoning about Tree Borrows. Kani [57] is an industry-grade, state-of-the-art symbolic testing tool for Rust, and it does so by compiling Rust code to the CBMC IL [30]. We compared Soteria$^{\text{Rust}}$ and Kani at length in §4.

Bi-abduction and Bug Finding. To our knowledge, there are two tools that perform bi-abductive SE for automatic bug finding: Infer [11, 32] and Gillian [22, 35, 38]. Infer is an industrial-strength tool by Meta that supports multiple languages. However, Infer is based on a single IL, SIL, leading to complex compilers. As such, extending Infer to a new language often requires modifying Infer itself by writing 'models' for some functions directly in OCaml. Gillian is the framework that is closest to our goal of adapting the tool to each language. Gillian allows one to override the notion of memory model directly in OCaml for each source language. However, Gillian still relies on an GIL, its IL, and each language instantiation must provide both a complex compiler to its IL and an OCaml implementation of all state operations. In addition, Gillian's expression language is fixed and captures constructs that must be compatible with both JavaScript and C, leading to a complex expression language that still requires constant decoding and re-encoding of values (which is our leading hypothesis for its performance compared to Soteria).

Data-Availability Statement

We provide the full implementation of Soteria described in this paper, as well as all the benchmarks and scripts used to reproduce the results in this paper in the accompanying artifact, available at https://doi.org/10.5281/zenodo.19856235 [6].

Acknowledgments

This project was sponsored by the Defense Advanced Research Projects Agency, (DARPA), Information Innovation Office (I2O), Program BAA HR001124S0003, under Cooperative Agreement No. HR00112420359. Disclaimer: The content of the information does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

Raad is further supported by the UKRI Future Leaders Fellowship MR/V024299/1, the EPSRC grant EP/X037029/1 and VeTSS. We would like to thank Guillaume Boisseau for his help with making Charon work for Soteria$^{\text{Rust}}$; Petar Maksimović for his feedback on the earlier drafts of this manuscript; the Cerberus developers for their help with setting up, using and improving Cerberus; Peter Sewell for hosting Ayoun as a visiting researcher at the University of Cambridge; and Peter O'Hearn for his guidance throughout this project. Finally, we thank the PLDI 2026 reviewers for their feedback and suggestions, which have greatly improved the quality of this paper.

References

[1] Léo Andrès, Filipe Marques, Arthur Carcano, Pierre Chambart, José Fragoso Santos, and Jean-Christophe Filliâtre. 2024. Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly. *The Art, Science, and Engineering of Programming* 9, 1 (Oct. 2024), 3:1–3:42. doi:10.22152/programming-journal.org/2025/9/3

[2] Flavio Ascari, Roberto Bruni, and Roberta Gori. 2024. Limits and Difficulties in the Design of Under-Approximation Abstract Domains. *ACM Trans. Program. Lang. Syst.* 46, 3 (Oct. 2024), 11:1–11:31. doi:10.1145/3666014

[3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 147:1–147:30. doi:10.1145/3360573

[4] Sacha-Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. 2025. A Hybrid Approach to Semi-automated Rust Verification. *Proc. ACM Program. Lang.* 9, PLDI, Article 186 (June 2025), 23 pages. doi:10.1145/3729289

[5] Sacha-Élie Ayoun. 2024. *Gillian: foundations, implementation, and applications of compositional symbolic execution.* Ph. D. Dissertation. doi:10.25560/119340

[6] Sacha-Élie Ayoun, Opale Sjöstedt, and Azalea Raad. 2026. *Artifact - Soteria: Efficient Symbolic Execution as a Functional Library.* doi:10.5281/zenodo.19856235

[7] Siddharth Bhat, Léo Stefanesco, Chris Hughes, and Tobias Grosser. 2025. Certified Decision Procedures for Width-Independent Bitvector Predicates. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 370 (Oct. 2025), 23 pages. doi:10.1145/3763148

[8] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. 2024. CaDiCaL 2.0. In *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14681)*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer, 133–152. doi:10.1007/978-3-031-65627-9_7

[9] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. 2024. CaDiCaL, Gimsatul, IsaSAT and Kissat Entering the SAT Competition 2024. In *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions (Department of Computer Science Report Series B, Vol. B-2024-1)*, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda (Eds.). University of Helsinki, 8–10.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, USA, 209–224.

[11] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, Berlin, Heidelberg, 459–465. doi:10.1007/978-3-642-20398-5_33

[12] Rust Community. 2025. Miri. https://github.com/rust-lang/miri

[13] Loïc Correnson. 2014. Qed. Computing What Remains to Be Proved. In *NASA Formal Methods*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Alfred Kobsa, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar

Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Demetri Terzopoulos, Doug Tygar, Gerhard Weikum, Julia M. Badger, and Kristin Yvonne Rozier (Eds.). Vol. 8430. Springer International Publishing, Cham, 215–229. doi:10.1007/978-3-319-06200-6_17

[14] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. doi:10.1145/512950.512973

[15] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. 2009. Why Does Astrée Scale Up? *Formal Methods in System Design* 35, 3 (Dec. 2009), 229–264. doi:10.1007/s10703-009-0089-6

[16] Daniel Cumming. [n. d.]. Introducing KMIR: Concrete and Symbolic Execution of Rust MIR. https://runtimeverification.com/blog/introducing-kmir.

[17] David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017), 12:1–12:25. doi:10.1145/3110256

[18] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[19] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *ICFEM 2022 - 23th International Conference on Formal Engineering Methods*. Springer Verlag.

[20] Jean-Christophe Filliâtre and Sylvain Conchon. 2006. Type-safe modular hash-consing. In *Proceedings of the 2006 Workshop on ML* (Portland, Oregon, USA) *(ML '06)*. Association for Computing Machinery, New York, NY, USA, 12–19. doi:10.1145/1159876.1159880

[21] Nima Rahimi Foroushaani and Bart Jacobs. 2022. Modular Formal Verification of Rust Programs with Unsafe Blocks. arXiv:2212.12976 [cs.LO] https://arxiv.org/abs/2212.12976

[22] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part i: A Multi-Language Platform for Symbolic Execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 927–942. doi:10.1145/3385412.3386014

[23] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. 2019. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 66:1–66:31. doi:10.1145/3290379

[24] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. [n. d.]. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. 8 ([n. d.]).

[25] Son Ho, Guillaume Boisseau, Lucas Franceschino, Yoann Prak, Aymeric Fromherz, and Jonathan Protzenko. 2025. Charon: An Analysis Framework for Rust. arXiv:2410.18042 [cs.PL] https://arxiv.org/abs/2410.18042

[26] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 116:711–116:741. doi:10.1145/3547647

[27] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. 2026. Miri: Practical Undefined Behavior Detection for Rust. *Proc. ACM Program. Lang.* 10, POPL, Article 48 (Jan. 2026), 29 pages. doi:10.1145/3776690

[28] Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. 2022. Verified Symbolic Execution with Kripke Specification Monads (and No Meta-Programming). *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 194–224. doi:10.1145/3547628

[29] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A Software Analysis Perspective. *Formal Aspects of Computing* 27, 3 (May 2015), 573–609. doi:10.1007/s00165-014-0326-7

[30] Daniel Kroening and Michael Tautschnig. 2014. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, Berlin, Heidelberg, 389–391. doi:10.1007/978-3-642-54862-8_26

[31] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs Using Linear Ghost Types. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 85:286–85:315. doi:10.1145/3586037

[32] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 81:1–81:27. doi:10.1145/3527325

[33] Nico Lehmann, Adam Geller, Niki Vazou, and Ranjit Jhala. 2022. Flux: Liquid Types for Rust. arXiv:2207.04034 [cs.PL] https://arxiv.org/abs/2207.04034

[34] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Report. INRIA.

[35] Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Caroline Cronjäger, Petar Maksimović, and Philippa Gardner. 2024. Compositional Symbolic Execution for Correctness and Incorrectness Reasoning. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:28. doi:10.4230/LIPIcs.ECOOP.2024.25

[36] Andreas Lööw, Seung Hoon Park, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Opale Sjöstedt, and Philippa Gardner. 2025. Compositional Symbolic Execution for the Next 700 Memory Models. *Compositional Symbolic Execution for the Next 700 Memory Models (Artefact)* 9, OOPSLA2 (Oct. 2025), 373:2815–373:2842. doi:10.1145/3763151

[37] Sirui Lu and Rastislav Bodík. 2023. Grisette: Symbolic Compilation as a Functional Programming Library. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 16:455–16:487. doi:10.1145/3571209

[38] Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 827–850. doi:10.1007/978-3-030-81688-9_38

[39] Kayvan Memarian. 2023. *The Cerberus C Semantics.* Technical Report UCAM-CL-TR-981. University of Cambridge, Computer Laboratory. doi:10.48456/tr-981

[40] Antoine Miné, Abdelraouf Ouadjaout, and Matthieu Journault. 2018. Design of a Modular Platform for Static Analysis. In *The Ninth Workshop on Tools for Automatic Program Analysis (TAPAS'18)*. Fribourg-en-Brisgau, Germany. https://hal.sorbonne-universite.fr/hal-01870001

[41] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2024. Easing Maintenance of Academic Static Analyzers. *International Journal on Software Tools for Technology Transfer* 26, 6 (Dec. 2024), 673–686. doi:10.1007/s10009-024-00770-1

[42] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, Berlin, Heidelberg, 41–62. doi:10.1007/978-3-662-49122-5_2

[43] Peter W. O'Hearn. 2019. Incorrectness Logic. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 10:1–10:32. doi:10.1145/3371078

[44] Gaurav Parthasarathy, Thibault Dardinier, Benjamin Bonneau, Peter Müller, and Alexander J. Summers. 2024. Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language. *Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language – Artifact* 8, PLDI (June 2024), 208:1510–208:1534. doi:10.1145/3656438

[45] Rust Language Project. 2025. *std::intrinsics — Compiler intrinsics module (Rust Standard Library).* https://doc.rust-lang.org/std/intrinsics/index.html Accessed: 2025-10-26.

[46] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 225–252. doi:10.1007/978-3-030-53291-8_14

[47] Grigore Roșu and Traian Florin Șerbănuță. 2010. An Overview of the K Semantic Framework. *The Journal of Logic and Algebraic Programming* 79, 6 (Aug. 2010), 397–434. doi:10.1016/j.jlap.2010.03.012

[48] Rust Project Developers. 2018. *hashbrown: A Rust port of Google's SwissTable hash map.* https://crates.io/crates/hashbrown Rust crate; MIT OR Apache-2.0 license.

[49] Rust Project Developers. 2026. *Crates.io most downloaded crates.* https://crates.io/crates?sort=downloads Accessed: 2026-03-23.

[50] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. 2018. Symbolic Execution for JavaScript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3236950.3236956

[51] The Kani Team. 2023. How Open Source Projects Are Using Kani to Write Better Software in Rust | AWS Open Source Blog. https://aws.amazon.com/blogs/opensource/how-open-source-projects-are-using-kani-to-write-better-software-in-rust/.

[52] The Racket Team. 2010. Reference: Racket. In *The Racket Language.* https://racket-lang.org/

[53] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. doi:10.1145/2509578.2509586

[54] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 530–541. doi:10.1145/2594291.2594340

[55] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. *SIGPLAN Not.*

50, 1 (Jan. 2015), 209–220. doi:10.1145/2775051.2676995

[56] Quinten van Woerkom. 2025. *finetime: Accurate, flexible, and efficient time keeping.* https://crates.io/crates/finetime Rust crate; MIT OR Apache-2.0 license.

[57] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. 2022. Verifying Dynamic Trait Objects in Rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22).* Association for Computing Machinery, New York, NY, USA, 321–330. doi:10.1145/3510457.3513031

[58] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. 2025. Tree Borrows. *Proc. ACM Program. Lang.* 9, PLDI, Article 188 (June 2025), 24 pages. doi:10.1145/3735592

[59] Sacha Élie Ayoun, Opale Sjöstedt, and Azalea Raad. 2026. Soteria: Efficient Symbolic Execution as a Functional Library (Extended Version). arXiv:2511.08729 [cs.PL] https://arxiv.org/abs/2511.08729

Received 2025-11-13; accepted 2026-04-03