



**HAL**  
open science

## Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly

Léo Andrès, Filipe Marques, Pierre Chambart, Arthur Carcano, José Fragoso Femenin dos Santos

### ► To cite this version:

Léo Andrès, Filipe Marques, Pierre Chambart, Arthur Carcano, José Fragoso Femenin dos Santos. Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly. 2024. hal-04627413v1

**HAL Id: hal-04627413**

**<https://hal.science/hal-04627413v1>**

Preprint submitted on 27 Jun 2024 (v1), last revised 25 Oct 2024 (v4)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly

Léo Andrès<sup>a,c</sup> , Filipe Marques<sup>b</sup> , Pierre Chambart<sup>a</sup> , Arthur Carcano<sup>a</sup> , and José Fragoso Santos<sup>b</sup> 

a OCamlPro SAS, 21 rue de Châtillon, 75014, Paris, France

b INESC-ID / Instituto Superior Tecnico, University of Lisbon, Rua Alves Redol 9, 1000-029, Lisbon, Portugal

c Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France

**Abstract** We presents Owi, an interpreter for WebAssembly written in OCaml. It can perform parallel symbolic execution thanks to its monadic interpreter and a multi-core choice monad. We show how we are getting good results comparing to the state of the art tools with only minimal efforts in terms of symbolic execution techniques. Our approach allows us to perform symbolic execution of C and Rust code and even of code mixing both languages.

## The Art, Science, and Engineering of Programming

---

Perspective The Engineering of Programming

Area of Submission Interpreters, virtual machines, and compilers, Testing and debugging



© Léo Andrès, Filipe Marques, Pierre Chambart, Arthur Carcano, and José Fragoso Santos  
This work is licensed under a “CC BY-NC 4.0” license  
Submitted to *The Art, Science, and Engineering of Programming*.

## 1 Introduction

WebAssembly [25] (Wasm) is a new binary compilation target designed to be the new standard for the Web. Wasm is currently supported by 99% of tracked browsers [48], enabling Web applications to compile to Wasm and benefit from its performance improvements. Beyond the Web, Wasm is extensively used to power server-side runtimes [1, 8, 15], IoT platforms [26], and blockchain smart contracts [49].

By allowing high-level programming languages such as C, C++, Rust, Java, Go, Haskell, and OCaml to compile to Wasm, opportunities arise for the introduction of new classes of bugs and vulnerabilities into the context of web applications. Issues in the original programs can be transposed to Wasm binaries via compilation [17], these include buffer overflows [33], format strings [5], and memory leaks [23]. By Leveraging these classical vulnerabilities, attackers can exploit Wasm binaries to perform attacks such as cross-site scripting (XSS), denial of service (DoS), and code injection. For this reason, it is crucial to ensure that the community has the necessary tools to analyse and verify the security of Wasm binaries.

Symbolic execution [30] is a powerful technique that allows for the exploration of all feasible paths of a program up to a bound, by running a program with symbolic inputs instead of concrete ones. It is a technique that has been applied with success to a variety of programming languages, including C [9, 22], Java [38], and JavaScript [21, 40, 41]. Currently, the state-of-the-art in symbolic execution of Wasm is embodied by tools such as WANA [49], Manticore [35], WASP [32], and Eunomia [27]. However, these tools either require complicated setups, need deep knowledge of the Wasm binary to be analysed, or scale poorly to large codebases [32].

We present Owi, a toolkit to work with Wasm within the OCaml ecosystem. Owi features a reference interpreter for Wasm capable of performing both concrete and symbolic execution. In this paper, we first describe how we designed reusable components and a modular interpreter from a concrete one, enabling the sharing of code between the concrete and symbolic interpreters (§3). Secondly, we explain how we use the choice monad to abstract over the type of values and to deal with branching in a way that is compatible with both concrete and symbolic execution (§4). Finally, we present Smt.ml, a multi back-end front-end library for SMT solvers in OCaml, and discuss how we integrated it with Owi to allow the use of different SMT solvers during symbolic execution (§5).

While our primary goal is to analyse standalone Wasm, we also demonstrate the potential for performing symbolic execution on other languages by compiling by compiling them to Wasm. We provide examples of symbolic execution applied to C and Rust code and describe our ongoing efforts to extend this functionality to support OCaml and other garbage-collected language by integrating WasmGC into Owi.

We evaluate Owi on the 2024 Test-Comp benchmarks [6]. We show that Owi is able to perform symbolic execution of C and obtain results comparable to the state-of-the-art tools such as KLEE [9], with shortcomings on a few benchmarks due to limitations in the current implementation of the memory model. Additionally, we also show that our multi-core choice monad allows Owi to reduce timeouts and increase bugs detected by 10% compared to a single-core implementation.

```

1 (module
2   (func $swap (param $x i32) (param $y i32)
3     (if (i32.gt_s (local.get $x) (local.get $y))
4       (then
5         (local.set $x
6           (i32.add (local.get $x) (local.get $y)))
7         (local.set $y
8           (i32.sub (local.get $x) (local.get $y)))
9         (local.set $x
10          (i32.sub (local.get $x) (local.get $y)))
11        (if (i32.gt_s
12          (i32.sub (local.get $x) (local.get $y))
13          (i32.const 0))
14          (then
15            unreachable
16          ))
17        ))
18   )
19 )

```

■ **Listing 1** Example Wasm program written in Wasm Textual Format (Wat).

**Contributions** In summary, our contributions are as follows: (1) Owi, a toolkit to work with Wasm within the OCaml ecosystem, featuring an abstract reference interpreter for Wasm (§3); (2) A robust and scalable symbolic execution engine for Wasm, built using our choice monad (§4); (3) A front end for testing C and Rust programs using our symbolic execution engine (§6).

## 2 Background

In this section we give a brief overview of Wasm, focusing on its syntax and semantics (§2.1), together with a high-level introduction to symbolic execution with a particular emphasis on the discipline followed in this paper (§2.2).

### 2.1 Wasm

WebAssembly [25, 39] is a low-level binary instruction format that offers compact representation, efficient validation and compilation, and ensures safe execution with minimal overhead. Wasm is not tied to any specific hardware, being language, hardware and platform-independent. Like other assembly languages, it is mainly used as a compilation target for high-level programming languages. Wasm is used in web browsers but also in cloud platforms or in standalone runtimes.

In the following we provide a brief overview of the Wasm version 1 syntax, which is supported by 99% of tracked browser at the time of writing [48]. A WebAssembly binary takes the form of a *module*. A Wasm module includes a collection of Wasm

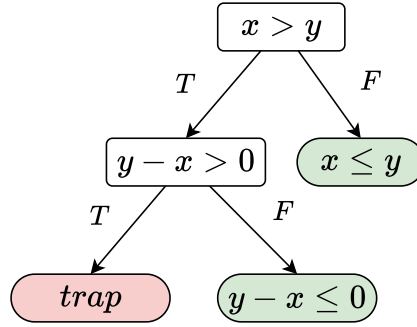
*functions*, together with the declaration of their shared global variables and the specification of a linear memory (a global array of bytes used as a heap). Computation is based on a *stack machine*; Wasm instructions interact with the stack by pushing values onto the stack or popping values out of the stack. A Wasm module is executed by an *embedder*, *i.e.* a host engine that defines how modules are loaded, but is also in charge of resolving imports and exports between modules, along with handling I/Os and traps.

The syntax of Wasm programs is exemplified in Listing 1 and includes: functions (e.g., `$swap`), local variables (e.g., `$x` and `$y`), values (e.g., `i32.const 0`), and instructions (e.g., `i32.add`, `local.get`, `if`, `unreachable`). Wasm has four *primitive types*: 32-bit integers (**i32**), 64-bit integers (**i64**), single-precision floating-point numbers (**f32**), and double-precision floating-point numbers (**f64**). Wasm makes no distinction between signed and unsigned *integers*, instead instructions have a *sign extension* to indicate how to interpret the generated integer values. *Wasm variables* can be either *local* or *global*. Local variables belong to the execution context of a function, whereas global variables belong to the module. Wasm does not have *named* variables; instead, both local and global variables are indexed by integers. The primary storage of a Wasm module is a large array of bytes, commonly referred to as *linear memory*. Memories can be imported or exported, meaning that they can be shared between modules. The initial size of a memory is fixed. However, memory can be programmatically grown when its size needs to be enlarged. In contrast to most stack machines, Wasm has structured control flow constructs, such as **if**, **else**, and **loop**. It ensures that humans can easily interpret Wasm code. For a comprehensive overview of the syntax and semantics of Wasm, the reader is referred to [25].

### 2.2 Symbolic Execution

Symbolic execution is a program analysis technique used to explore all feasible paths of a program up to a bound [30]. Instead of running a program using concrete inputs, symbolic execution engines run the given program with *symbolic* inputs. Every time the symbolic execution engine encounters a conditional branch point with a symbolic guard, it forks the current execution to be able to explore both branches. For each path of the program execution tree, the symbolic execution engine builds a logical formula, called the *path condition*, that represents the constraints on the inputs that must be satisfied in order to reach that path. In particular, every time a conditional branch point is symbolically executed, the current path condition is extended with its guard in the *then* branch and its negation in the *else* branch. Symbolic execution engines rely on an underlying SMT solver to check the feasibility of execution paths and the validity of assertions supplied by the developer. An execution path is said to be *feasible* if it can be realised by at least one concrete path and an assertion holds at a given point in the program if it is implied by the path condition at that point.

**Symbolic Execution Example** Let us now take a look at how symbolic execution works in practice. Consider the `$swap` function in the Wasm program given in Listing 1. This function is annotated with a final `unreachable` instruction, which is supposed



■ **Figure 1** Execution tree of the `$swap` function.

to never be executed, whatever input the program is given. In order to verify this, one has to explore all feasible execution paths of the function, which we illustrate in Figure 1 in the form of an execution tree. In the tree, we depict in green the leaf nodes corresponding to execution paths for a valid and desirable outcome of the function and in red those corresponding to paths for which the execution traps with an unreachable instruction. The trap is raised for the left-most path. To see this, consider the inputs  $x = 8388481$  and  $y = -2147483648$ . These inputs cause 32-bit integers to overflow when performing the subtraction  $x - y$  in the second branch of the function (*i.e.*  $-2147483648 - 8388481 \bmod 2^{32} = 2139095167$ ), leading to the trap. Below, we explain how these inputs can be discovered.

Since there are three possible execution paths in the `$swap` function, each of these needs to be symbolically executed. We will assume a breadth-first exploration strategy in this example. When symbolic execution starts on function `$swap`, the local variables `$x` and `$y` are assigned symbolic values, say  $x$  and  $y$ , respectively. As the first conditional branch is executed, the condition and its negation are both satisfiable; therefore, execution forks into two paths: one where  $x \geq y$  and one where  $x < y$ , with the path condition is updated accordingly:

$$PC_T \equiv x \geq y \quad \text{and} \quad PC_F \equiv x < y$$

Execution proceeds down  $PC_T$ . In this path the values of `$x` and `$y` are swapped, with `$x` taking the value of `$y` and vice versa. Next, the second conditional branch (*i.e.* `$x - $y > 0`) is executed, generating two new path conditions:

$$\$x \mapsto y \quad \$y \mapsto x \quad PC_{TT} \equiv x \geq y \wedge y - x > 0 \quad \text{and} \quad PC_{TF} \equiv x \geq y \quad y - x \leq 0$$

Both path conditions are satisfiable, leading to another fork in the execution. Due to our assumption of breadth-first exploration, the next path to be explored would be  $PC_F$ . This path corresponds to the execution of the *else* branch of the first conditional statement. In this path there is nothing more to do, and the execution terminates. So, we proceed with the next path that is described by  $PC_{TT}$ . In this path, the execution

reaches the `unreachable` instruction. One can ask the SMT for a model of the path condition  $PC_{TT}$ , which will provide the concrete values that lead to the trap:

$$\$x \mapsto 8388481 \quad \$y \mapsto -2147483648$$

### 3 From the Concrete to the Symbolic Interpreter

Owi's source code is available online [2]. Originally, Owi was a classical, concrete interpreter for Wasm written in OCaml. In this section, we explain how we turned it into a monadic interpreter. This transformation allows Owi to perform parallel symbolic execution in addition to concrete execution while preserving most of the original code.

#### 3.1 The Interpreter

The code of the original interpreter looked like this:

```
1 module Interpreter = struct
2
3   let rec eval_instr i stack =
4     match i, stack with
5     | Binop Add, Int32 x :: Int32 y :: stack ->
6       (Int32.add x y) :: stack
7     | Binop GT, Int32 x :: Int32 y :: stack ->
8       (Int32.gt x y) :: stack
9     | If_else (if_top, if_bot), cond :: stack ->
10      if cond then eval_expr if_top stack
11              else eval_expr if_bot stack
12      (* ... *)
13   and eval_expr e stack = (* ... *)
14
15 end
```

It was operating on an `Int32` OCaml module manipulating scalar values of type `int32`. The first thing we want to do is to abstract over the type of `Int32` and other values. To do this, we define a signature for values:

```
1 module type Values = sig
2   module Int32 : sig
3     type t
4     val add : t -> t -> t
5     val gt : t -> t -> t
6   end
7   (* ... *)
8 end
```

Then we functorize the `Interpreter` module by adding a `Values` module parameter of signature `Values`:

```

1 module Interpreter (Values : Values) = struct
2
3   open Values
4
5   let rec eval_instr i stack =
6     match i, stack with
7     | Binop Add, Int32 x :: Int32 y :: stack ->
8       (Int32.add x y) :: stack
9     | Binop GT, Int32 x :: Int32 y :: stack ->
10      (Int32.gt x y) :: stack
11     | If_else (if_top, if_bot), cond :: stack ->
12       if cond then eval_expr if_top stack
13         else eval_expr if_bot stack
14     (* ... *)
15   and eval_expr e stack = (* ... *)
16
17 end

```

In the concrete case, the `Values.Int32` module will be our previous `Int32` module. In the symbolic case, it will be a module operating on *expressions* similar to those that the path condition is made of.

Then, in order to perform concrete and symbolic execution while still sharing the code, we are going to use the *choice monad*. This module has the following signature:

```

1 module type Choice = sig
2   type 'a t
3   val return : 'a -> 'a t
4   val bind : 'a t -> ('a -> 'b t) -> 'b t
5   val select : Value.Bool.t -> bool t
6 end

```

Then we functorize the interpreter by a module having this signature, define some helpers and rewrite the `If_else` case:

```

1 module Interpreter (Values : Values) (Choice : Choice) = struct
2
3   open Values
4
5   let ( let* ) = Choice.bind
6
7   let rec eval_instr i stack =
8     match i, stack with
9     | Binop Add, Int32 x :: Int32 y :: stack ->
10      Choice.return @@ (Int32.add x y) :: stack
11     | Binop GT, Int32 x :: Int32 y :: stack ->
12      Choice.return @@ (Int32.gt x y) :: stack
13     | If_else (if_top, if_bot), cond :: stack ->
14      let* cond = Choice.select cond in
15      if cond then eval_expr if_top stack
16        else eval_expr if_bot stack

```



```
17   (* ... *)
18   and eval_expr e stack = (* ... *)
19
20 end
```

In the concrete case, the implementation of Choice is the identity:

```
1 type 'a t = 'a
2
3 let return x = x [@@inline]
4
5 let bind x f = f x [@@inline]
6
7 let select b = b [@@inline]
```

The code is going to behave in the exact same way and there is no abstraction runtime cost thanks to the right `[@@inline]` annotations and the OCaml compiler with the Flambda optimizer enabled.

In the symbolic case, the choice monad is actually the continuation (or coroutine) monad mixed with the state monad. The state will store the path condition. The select function will add the correct expressions to the path condition and call an SMT solver to determine which branches are reachable. When both branches are reachable, we will interpret both of them and part of the state needs to be duplicated. This may not be apparent in the code, as it is managed by the `let*` that calls bind. Finally, the branches are explored in parallel using the multicore capabilities of OCaml 5. The implementation of the monad is detailed in the next section.

This monadic functorization allowed us to share most of the code between both interpreters, while maintaining good performances and with only minimal changes required to the original code. We believe this approach to be a good way to easily get a symbolic interpreter from a concrete one in other settings.

## 4 The Choice Monad

In this section, we present the choice monad. We first describe the choice monad and explain how it powers symbolic execution (§4.1). Then, we briefly discuss the lazy memory model used to prevent memory over-consumption (§4.2).

### 4.1 Multicore Implementation

During symbolic execution of a program, the symbolic interpreter is often faced with a choice, say deciding whether a Boolean is true or false. Each choice corresponds to a different branch of the DAG of possible executions, and each of these branches must be explored. This exploration is ideally done in parallel. We have highlighted how the monadic construction of our interpreter allowed us to turn a concrete interpreter to a symbolic one. In this section, we will detail how our symbolic execution monad allows for parallel execution of the code.

Our monad composes together several functionalities:

- Those of a forkable coroutine monad. That is the execution can *yield* back to its executor and *fork* (*i.e.* duplicate) itself. Moreover these coroutines have access to a *worker-local* storage: that is some value that is owned by the worker currently executing the coroutine.
- Those of a state monad. The state considered here being the interpreter’s internal “Wasm” state.
- Those of an error monad. Wasm programs can trigger traps and Owi extends Wasm with assertions that can fail. Both these cases need to be propagated up.

This composition is done using three layered monad transformers. For the sake of brevity and simplicity these transformers are implemented inline rather than as module functors.

The state and error monad are similar to those described in the literature [28]. The coroutine monad  $\alpha \rightarrow (\alpha, wls) t$  for a worker-local storage type *wls* is implemented like so in OCaml:

```
1 type ('a, 'wls) t =  
2   Sched of ('wls -> ('a, 'wls) status)  
3  
4 and ('a, 'wls) status =  
5   | Now of 'a  
6   | Yield of Prio.t * ('a, 'wls) t  
7   | Choice of (('a, 'wls) status * ('a, 'wls) status)  
8   | Stop
```

Elements of the monad read a worker local storage (WLS) and use it to compute a return value that is either:

- Now denoting a final value;
- Yield which gives both a priority for the next computation step and a new element of the monad that is the next step in the computation pipeline;
- Choice which indicates the creation of a new coroutine;
- Stop which indicates that the execution finishes without any value.

This representation of coroutines needs an accompanying scheduler. Our current implementation uses two queues. One for the remaining work (to which we will push and from which we will read), the other (to which we will only push) for results (Now values). Each of the scheduler’s worker alternatively pop the highest priority routine from the work queue, execute it with its local storage, and handle its result, repushing returned routines to the work queue and writing results (Now values) to the result queue.

From this basic blocks we can write a `yield` primitive which gives control back to the scheduler, and a `choose` function which takes two monadic values and combines them into one, while taking care of appropriately cloning mutable state for the two children. We can also write an `add_pc` which adds a new hypothesis to the current path condition.

Using those, we can finally write our `select`:

## Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly

```
1 let check_reachability =
2   let* () = yield in
3   match Solver.check s thread.pc with
4   | `Sat -> return ()
5   | `Unsat | `Unknown -> stop
6
7 let select (v : Symbolic_value.bool) =
8   let top_branch =
9     let* () = add_pc v in
10    let+ () = check_reachability in
11    true
12  in
13  let bot_branch =
14    let* () = add_pc (Symbolic_value.Bool.not v) in
15    let+ () = check_reachability in
16    false
17  in
18  choose top_branch bot_branch
```

Here we yield before each call to the SMT solver.

### 4.2 Lazy Memory Model

A classic problem with symbolic execution is the potential for excessive memory consumption due to the path explosion problem [3, 10], even in small programs. When we ran Owi on the Test-Comp benchmarks [6] using a 30 second timeout, we observed a maximum duplication of 156,052 states. If we were to adopt a naive approach of duplicating the state each time a branch is taken, it would require storing 156,052 copies of the Wasm linear memory, which typically varies from 64KiB to 4GiB in size. Making it unreasonable to store all these copies in memory. This rapid state explosion highlights the critical importance of efficient state management.

Owi uses a lazy memory model akin to a copy-on-write strategy. When a new state is created, it does not duplicate the memory. Instead, it creates a new memory that points to the same memory as the original state, along with a map that stores the modifications. When accessing a memory location, Owi first checks if the location is present in the modifications map. If it is, the value is returned; otherwise, the value is fetched from the original memory. This process is recursive, as the original memory may itself be a modification map. Writes are always performed in the modifications map

This strategy is similar to the approach used by KLEE [9]. However, KLEE models memory as a map of objects, which requires copying an entire object to the modifications map even for small changes. In contrast, Owi models memory as a large array of bytes, allowing it to copy only the modified bytes. This is more efficient in terms of memory usage.

```

1 type t = expr hash_consded
2 and  expr =
3   | Val of Value.t
4   | Unop of unop * t
5   | Binop of binop * t * t
6   | Triop of triop * t * t * t
7   | Relop of relop * t * t
8   | Cvtop of cvtop * t
9   | Symbol of Symbol.t
10  | Extract of t * int * int
11  | Concat of t * t

```

■ Listing 2 Abstract Syntax of Smt.ml.

## 5 Interacting With SMT Solvers

Smt.ml is an OCaml SMT abstraction layer for constraint solvers. The primary objective of Smt.ml is to facilitate the effortless transition between different SMT solvers during program analysis, as certain SMT solvers may prove more efficient at handling specific logics and formulas. Presently, Smt.ml offers support for Z3 [19], Colibri2 [45], and Bitwuzla [36], and ongoing efforts are directed towards incorporating support for Alt-Ergo [16] and cvc5 [4].

**Abstract Syntax** The principal means of interacting with `smtml` is through its abstract grammar. To begin, logical terms must be encoded as `smtml` expressions for subsequent satisfiability checks using one of the available solver backends. The abstract grammar is defined as the algebraic datatype in Listing 2.

The grammar includes representations for concrete values, unary and binary operations, ternary operations, relational operations, conversion operations, symbols, and specific operations such as extraction and concatenation. Allowing for the concise representation of logical terms, such as:

```
Relop (Ge, Symbol "x", Val (Int 42))
```

In this example, the abstract grammar is employed to express the logical constraint ( $x \geq 42$ ). The `Relop` constructor signifies a relational operation, specifically the Greater Than or Equal (`Ge`) operator. The `Symbol` constructor represents the variable “x”, and the `Val` constructor denotes the concrete value, here, an integer with the value 42. Smt.ml also provides smart constructors to allow building *hash-consed* terms:

```
relop Ge (symbol "x") (int 42)
```

**Parametric Solvers** Smt.ml provides a succinct manner for interacting with SMT solvers, as independently of what solver one chooses, the signature of the `Solver` module is always the same. This module, offers a consistent set of functions facilitating seamless integration with SMT solvers. These include `check`, `push`, `pop`, `model`, and `get_value`.

## Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly

To provide easy integration with multiple SMT solvers we functorize the solver module with module with a well-defined set of functions provided by each SMT solver:

```
module Make (M : Prover_mappings_intf.S) = struct ...
```

This functor, denoted as `Make`, takes a module `M` as an argument. The module `M` should adhere to the signature defined in `Prover_mappings_intf.S`, outlining a standard set of functions expected from each SMT solver integration.

**Integration with Owi** Thanks to Owi's functorized interpreter, integrating `Smt.ml` is a seamless process. It requires providing a `values` module that adheres to the interpreter's expected interface. Within this module, expressions are constructed using `Smt.ml`'s smart constructors. Referring back to the `Values` mode from Section 3, the symbolic execution module now takes the following form:

```
1 open Smtml (* Import the Smt.ml module *)
2 module Values = struct
3   module Int32 =
4     type t = Expr.t
5     let v i = Expr.value (I32 i)
6     let add lhs rhs = Expr.relop Add lhs rhs
7     let gt lhs rhs = Expr.relop Gt lhs rhs
8   end
9   (* ... *)
10 end
```

## 6 Symbolic Execution of C and Rust Programs

We have showcased Owi's ability to find bugs in Wasm codebases. However, many languages can now also be compiled to Wasm. By exposing Owi's API to those host languages, we can hence use Owi itself to identify bugs in any program that can be compiled to Wasm.

To use Owi to analyze a program in a language `L` one must:

- Implement the `L` primitives to interact with the program environment (disk and network IO, `syscall`) in a way that correctly models them for Owi analysis. This includes modeling dynamic memory allocation (for example `malloc`, `realloc`, and `free` in C).
- Bind Owi primitives to generate symbols, assert properties and stop exploration in `L`.

Once this work done, we can compile a `L` program to Wasm, and run it in Owi to identify bugs. However, one must be aware that some bugs, and especially those relating to undefined behavior, can be masked by the compiler. Our Wasm program is only one interpretation of the `L` program, which can have several if it contains undefined behavior.

We detail our work done on C and Rust in the following subsections.

## 6.1 C

We have implemented part of the C standard library, as well as a C header file allowing to interact with Owi from C. We have also added a subcommand to our Owi binary to drive the compilation to Wasm of C programs. As illustrated in the following code examples, we can now use our bindings to test C programs.

```

1 #include <owi.h>
2
3 int main(void) {
4     int x = owi_i32();
5     int y = owi_i32();
6
7     if (x > y) {
8         x = x + y;
9         y = x - y;
10        x = x - y;
11        if (x - y > 0) {
12            owi_assert(0);
13        }
14    }
15 }

```

```

1 $ owi c swap.c
2 Assert failure: false
3 Model:
4   (model
5     (symbol_0 (i32 8388481))
6     (symbol_1 (i32 -2147483648)))
7 Reached problem!

```

## 6.2 Rust

A similar work has been done in Rust. We have not so far modeled the standard library, but have demonstrated that Owi can be used to check that rewriting a C function to Rust preserves its semantic.

For example consider the following C function, that computes the average of two numbers while correctly handling overflow.

```

1 #include <stdint.h>
2
3 int32_t mean_c(int32_t a, int32_t b) {
4     return (a & b) + ((a ^ b) >> 1);
5 }

```

and its tentative Rust rewrite, by a programmer unaware of overflows

## Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly

```
1 fn mean_rust(x: i32, y: i32) -> i32 {
2     (x + y) / 2
3 }
```

The following main checks if both functions have the same semantic for all inputs.

```
1 fn main() {
2     let x = owi_sym::u32_symbol() as i32;
3     let y = owi_sym::u32_symbol() as i32;
4     owi_sym::assert(mean_c(x, y) == mean_rust(x, y))
5 }
```

After proper compilation and linking (not displayed here), we get that

```
1 $ owi sym target/wasm32-unknown-unknown/release/rust-owi-opt.wat
2 Assert failure: (bool.eq (i32.add (i32.and symbol_0 symbol_1) (i32.shr
  ↳ (i32.xor symbol_0 symbol_1) (i32 1))) (i32.div (i32.add symbol_0
  ↳ symbol_1) (i32 2)))
3 Model:
4 (model
5   (symbol_0 (i32 -2147483520))
6   (symbol_1 (i32 -2147483519)))
```

Fixing the Rust function for

```
1 fn mean_rust(x: i32, y: i32) -> i32 {
2     (x & y) + ((x ^ y) >> 1)
3 }
```

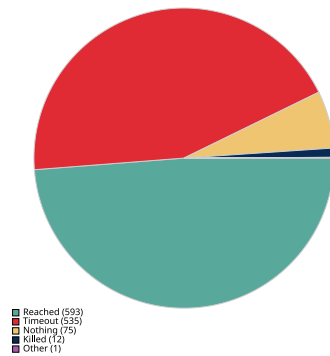
we finally get

```
1 $ owi sym target/wasm32-unknown-unknown/release/rust-owi-opt.wat
2 All OK
```

## 7 Experimental Evaluation

In order to measure the performance of our symbolic interpreter, we are using the benchmarks from Test-Comp [6] that are available in the sv-benchmarks repository [7]. In the 2024 edition, it was composed of two categories. We are interested in the Cover-Error one. It is made of 1217 tasks, each one being a C file in which there is one bug that should be found. We are running each task with a timeout set to 30 seconds.

Unless specified otherwise, all our benchmarks are running on a AMD EPYC 7451 24-Core Processor server, which has 48 threads and 128G of RAM. Owi is compiled with the OCaml 5.2.0 compiler with the Flambda1 optimizer option enabled. We are running Owi with the -w24 flags which sets the number of workers for multicore execution to 24 and -O3 which tells clang which code optimisation level it should apply. To compile C code to Wasm in Owi we used the version 14 of LLVM, which is the



■ **Figure 2** Score of Owi on Test-Comp.

one that was available on the server. Moreover, all the code to run our benchmarks in a reproducible fashion, generate our diagrams and compare benchmarks across tools and various parameters is available in the repository of Owi.

### 7.1 Score and Distribution of Execution Times

In Figure 2, we observe that Owi correctly detected the bug for 593 tasks out of 1216 (we disabled one of them that contains inline X86 assembly code). It reached the timeout limit for 535 of them. For 75 of them, it did not report anything, these cases are due to some limitations in our current memory model; when such a limitation is found Owi simply stops exploring the current path and try the others — there’s an option to emit a warning when such a limitation is reached. The others cases are due to issues while compiling the C code to Wasm or runtime errors such as being out-of-memory killed.

In Figure 3, we can see the distribution of execution times. It appears that most problem are solved by Owi in a very short time, more than half of the problems in the *reached* category are solved in less than a second.

### 7.2 Trying Different Parameters

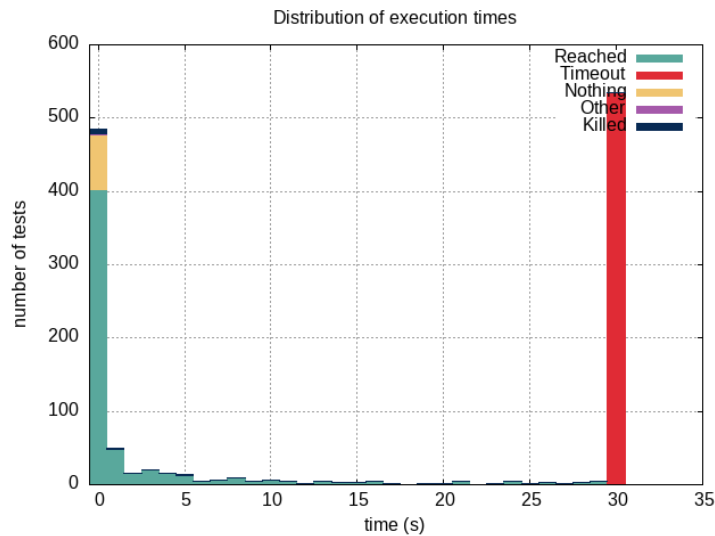
**Multicore Evaluation.** In order to measure the efficiency of the multi-core implementation, we ran Owi with various number of workers, from `-w1` to `-w48`. Here the benchmarks were all done with the `-O1` optimisation level flag.

The results in Table 1 are showing that the number of reached tasks is growing with the number of workers, until the optimal number which is 24. Above the optimal number, it slowly starts to decrease, probably due to an increased contention. Comparing the runs with 1 and 24 workers shows that we are solving 46 more tasks thanks to the multi-core version, which is almost a ten percent improvement.

Increasing the number of workers also lead to more killed runs, this is probably because we have more out-of-memory errors although we did not investigated properly



## Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly



■ **Figure 3** Distribution of execution times of Owi on Test-Comp.

Number of workers	Reached	Timeout	Nothing	Other	Killed
1	545	593	75	1	2
16	589	542	75	1	9
24	591	540	75	1	9
32	588	540	75	1	12
48	581	543	75	1	16

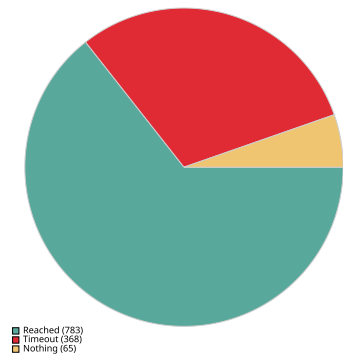
■ **Table 1** Results of Owi on Test-Comp with various number of workers.

for now. The distribution of execution times is quite similar in all cases so we are not displaying them here.

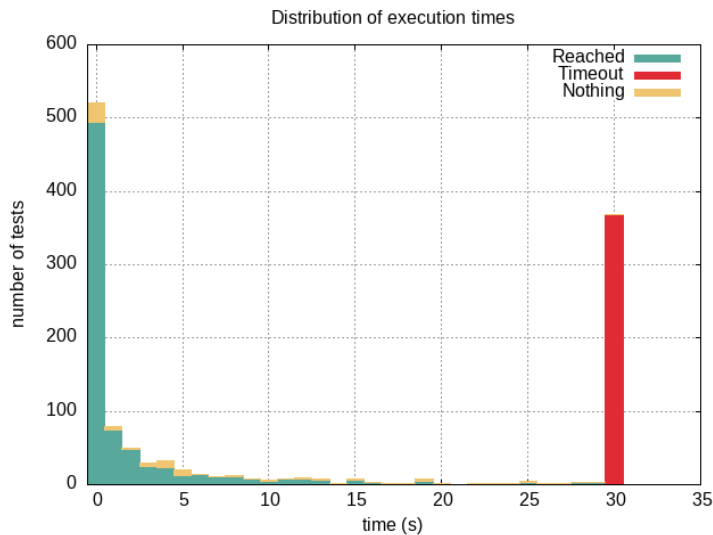
**Clang Optimisations.** We also tried using the options `-O0` and `-O3` to see if they were making any difference. In the `-O0` case Owi is solving only 462 tasks. It seems all examples that are not solved anymore are related to some loops not being optimised anymore and leading Owi to be stuck in them while doing the exploration. In the `-O1` case, Owi is solving 591 tasks and 593 in the `-O3` case. This is a great improvement compared to `-O0` and this suggests that most optimisations designed for concrete execution are also good optimisations for symbolic execution, even when taking the increased compilation time into account.

### 7.3 Comparing Against KLEE

KLEE [11] is one of the state-of-the-art tool in bug findings, it ranked 2nd on the 2024 edition of Test-Comp. We ran it on the same server and with the same timeout



■ **Figure 4** Score of KLEE on Test-Comp.



■ **Figure 5** Distribution of execution times of KLEE on Test-Comp.

as Owi in order to get comparable results (we re-used the artefact sent to Test-Comp to make sure we were running it properly). Here are the results:

From Figure 4 we notice that KLEE is performs better than Owi, detecting the bug in 783 tasks. We made a proper comparison analysis of the results of the two tools. Our findings are as follow. Among the tasks that were solved, 556 are reached by both solvers. There are 227 tasks solved by KLEE and not by Owi; but also 37 tasks solved only by Owi.

The mean wall-clock execution time on the tasks reached by both tools is 1.43 seconds for KLEE and 2.92 for Owi. The distribution of execution times in Figure 5 is quite similar to Owi.

For tasks solved only by Owi, KLEE had the following results: 10 “nothing” and 27 timeouts. Similarly, for tasks solved by KLEE only, Owi replied: 70 nothing, 145

timeout and 12 other or killed. This shows that improving the Owi memory model to handle the missing cases would already allows Owi to find one third of the bugs it is currently missing. We are not able to tell why Owi is performing better than Klee on some of the cases as it requires a deep understanding of the various techniques used by KLEE.

### 8 Related Work

The goal of this work is to provide a scalable and maintainable Wasm interpreter, capable of performing symbolic execution efficiently. Closely related to our work are: monadic and parametric interpreters, and symbolic execution tools for Wasm.

**Monadic and Parametric Interpreters** Our work is closely related to [34], which derives a symbolic execution engine from a definitional interpreter. Similar to our Wasm interpreter, the authors start from a concrete implementation and derive a parametric one with an interface of abstract values. They then, instantiate the interpreter in two different ways to obtain a concrete and a symbolic interpreter. The main difference with our work is that they focus on dynamically-typed languages with recursive functions and pattern matching. Additionally, in the symbolic instance of their interpreter, they choose a breadth-first search strategy to explore the state space of the program. In contrast, our use of a co-routine monad allows us to prioritise the exploration of more relevant branches.

Necro [37] is a framework to formalise programming-language semantics which allows for the automatic generation of interpreters or proofs in Coq [46]. Given the semantics of a language, Necro will generate an OCaml functor parametric on a interpretation monad that deals with applications and branches. This approach is similar to ours in that it leverages monads to manage computation effects. However, Necro focuses on concrete interpretation and does not extend to abstract interpretation.

**Symbolic Execution for Wasm** Symbolic execution has been extensively used to find crucial errors and vulnerabilities in a broad spectrum of programming languages, such as C [22], C++ [9], Java [43], and Python [14]. Regarding the Web, there are several state-of-the-art tools for symbolically executing JavaScript code [31, 40, 41, 42, 44], demonstrating the need for such tools for the validation and testing of modern Web applications.

Symbolic execution tools can be divided into two main classes: *static* and *dynamic/concolic* [3]. Static symbolic execution engines, such as [29, 30, 38, 40, 41, 47], explore the entire symbolic execution tree up to a pre-established depth, while concolic execution engines, such as [9, 13, 22, 31, 42, 43, 44], usually work by pairing up a concrete execution with a symbolic execution and exploring one execution path at a time. There is a vast body of research on both static and concolic symbolic execution tools for a wide variety of programming languages, see [3, 10, 12] for comprehensive surveys on the topic. In the following, we give a detailed account of the existing symbolic execution tools for Wasm other than Owi.

WANA [49] is a cross-platform tool for detecting vulnerabilities in smart contracts using static symbolic execution over Wasm bytecode. It employs specific heuristics for each platform but lacks a stand-alone symbolic execution engine for arbitrary Wasm code. Manticore [35] is a flexible symbolic execution framework for binaries and smart contracts, including Wasm bytecode, but requires complex, manually written Python scripts for each test and is no longer maintained. WASP [32], built on the Wasm reference interpreter, uses concolic execution to reduce solver interactions and simplify memory modeling, but its single-threaded architecture limits scalability and makes maintenance challenging. Eunomia [27] employs a novel symbolic execution technique with fine-grained local search strategies defined in a DSL, but its effectiveness is limited by its reliance on an outdated Wasm decoder, and the need for deep program understanding to leverage local strategies effectively.

None of these tools [9, 27, 32, 35, 49] are capable of exploring the state space of a program in a parallel or concurrent manner. Our work, is the first symbolic execution engine for Wasm that explores the state space of a program in parallel, leveraging OCaml-Multicore.

## 9 Conclusion and Perspectives

Now that Owi is able to perform parallel symbolic execution with good results, there are many things we would like to try in order to improve its bug-finding abilities. The first of them is to complete the memory model in order to be able to handle all cases which are currently returning “nothing”. We also implemented a concolic version of Owi, which is still sharing the code of the concrete and symbolic interpreters - it is already available and works and small examples but we haven’t been able to benchmark it properly due to some issue related to the memory handling. We would like to implement an efficient exploration strategy to take advantage of the notion of priority allowed by our choice monad, for instance with  $A^*$  [18]. We have some ideas to avoid the combinatorial explosion of path numbers (especially in the presence of loops) by using constrained Horn clauses (CHC) [24] to infer loop invariants and by using hash-consed Patricia trees [20] to merge states efficiently. Last but not least, we would like to implement the WasmGC proposal in Owi, in order to be able to perform symbolic execution of OCaml code by compiling it with Wasocaml (our OCaml to WasmGC compiler).

As a conclusion, we first believe our work demonstrate that Wasm is a good fit for symbolic execution. Indeed, more and more languages are targeting Wasm and it’s quite easy to re-use the tool-chain provided by each language to compile to Wasm and perform symbolic execution on the generated code. It also allows to perform cross-language symbolic execution. Moreover, we give a re-usable technique to implement performant symbolic execution easily, by making a concrete interpreter monadic and using a clean choice monad implementation. This technique is easy to apply in other settings and lead to quite decent performance as showed in our evaluation. We are confident about the fact that adding some more advanced symbolic execution techniques on top of Owi will lead to very good results.

## Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly

**Acknowledgements** Thanks to Jean-Christophe Filiâtre for all the relevant feedback on the article.

## References

- [1] Syrus Akbary and Ivan Enderlin. *Wasmer: Run any code on any client*. Accessed 27th-October-2021. URL: <https://wasmer.io>.
- [2] Léo Andrès, Pierre Chambart, Filipe Marques, Eric Patrizio, and Arthur Carcano. *Wasocaml*. 2021. URL: <https://github.com/ocamlpro/owi>.
- [3] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. “A Survey of Symbolic Execution Techniques”. In: *ACM Computing Surveys* (2018).
- [4] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. “cvc5: A versatile and industrial-strength SMT solver”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2022, pages 415–442.
- [5] John Bergbom. *Memory safety: old vulnerabilities become new with WebAssembly*. Technical report. Forcepoint, Dec. 2018.
- [6] Dirk Beyer. “Status Report on Software Testing: Test-Comp 2021”. In: *Fundamental Approaches to Software Engineering*. 2021.
- [7] Dirk Beyer. *SV-Benchmarks*. Accessed 28th-October-2021. URL: <https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks>.
- [8] Ruben Bridgewater. *Node v12.3.0*. Accessed 27th-April-2024. URL: <https://nodejs.org/en/blog/release/v12.3.0>.
- [9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *USENIX Conference on Operating Systems Design and Implementation*. 2008.
- [10] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. “Symbolic execution for software testing in practice: preliminary assessment”. In: *International Conference on Software Engineering*. 2011.
- [11] Cristian Cadar and Martin Nowack. “KLEE symbolic execution engine in 2019”. In: *International Journal on Software Tools for Technology Transfer* 23 (2021), pages 867–870.
- [12] Cristian Cadar and Koushik Sen. “Symbolic Execution for Software Testing: Three Decades Later”. In: *Communications of the ACM* (2013).
- [13] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. “Unleashing Mayhem on Binary Code”. In: *IEEE Symposium on Security and Privacy*. 2012.

- [14] Ting Chen, Xiao-song Zhang, Rui-dong Chen, Bo Yang, and Yang Bai. “Conpy: Concolic Execution Engine for Python Applications”. In: *Algorithms and Architectures for Parallel Processing*. 2014.
- [15] Lin Clark. *Standardizing WASI: A system interface to run WebAssembly outside the web*. Accessed 27th-October-2021. URL: <https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface>.
- [16] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. “Alt-Ergo 2.2”. In: *SMT Workshop: International Workshop on Satisfiability Modulo Theories*. 2018.
- [17] Michael Pradel Daniel Lehmann Johannes Kinder. “Everything Old is New Again: Binary Security of WebAssembly”. In: *USENIX Security Symposium*. 2020.
- [18] Theo De Castro Pinto, Antoine Rollet, Grégoire Sutre, and Ireneusz Tobor. “Guiding Symbolic Execution with A-Star”. In: *Software Engineering and Formal Methods*. Edited by Carla Ferreira and Tim A. C. Willemse. Cham: Springer Nature Switzerland, 2023, pages 47–65. ISBN: 978-3-031-47115-5.
- [19] Leonardo De Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2008.
- [20] Jean-Christophe Filliâtre and Sylvain Conchon. “Type-safe modular hash-consing”. In: *Proceedings of the 2006 Workshop on ML*. 2006, pages 12–19.
- [21] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. “Gillian, Part I: A Multi-Language Platform for Symbolic Execution”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020.
- [22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: Directed Automated Random Testing”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2005.
- [23] GoogleSecurityResearch. *Google Chrome 73.0.3683.103 - 'WasmMemoryObject::Grow' Use-After-Free*. Accessed 27th-October-2021. URL: <https://www.exploit-db.com/exploits/46968>.
- [24] Arie Gurfinkel and Nikolaj Bjørner. “The science, art, and magic of constrained horn clauses”. In: *2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE. 2019, pages 6–10.
- [25] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. “Bringing the web up to speed with WebAssembly”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017.
- [26] Adam Hall and Umakishore Ramachandran. “An Execution Model for Serverless Functions at the Edge”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2019.

## Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly


- [27] Ningyu He, Zhehao Zhao, Jikai Wang, Yubin Hu, Shengjian Guo, Haoyu Wang, Guangtai Liang, Ding Li, Xiangqun Chen, and Yao Guo. “Eunomia: Enabling User-Specified Fine-Grained Search in Symbolically Executing WebAssembly Binaries”. In: *ISSTA*. 2023.
- [28] Mark P Jones and Luc Duponcheel. *Composing monads*. Technical report. Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale ..., 1993.
- [29] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. “Generalized Symbolic Execution for Model Checking and Testing”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2003.
- [30] James C. King. “Symbolic Execution and Program Testing”. In: *Communications of the ACM* (1976).
- [31] Guodong Li, Esben Andreasen, and Indradeep Ghosh. “SymJS: Automatic Symbolic Testing of JavaScript Web Applications”. In: *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014.
- [32] Filipe Marques, José Fragoso Santos, Nuno Santos, and Pedro Adão. “Concolic Execution for WebAssembly”. In: *ECOOP*. 2022.
- [33] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. *Security Chasms of WASM*. Technical report. NCC Group, Aug. 2018.
- [34] Adrian D. Mensing, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. “From definitional interpreter to symbolic executor”. In: *International Workshop on Meta-Programming Techniques and Reflection*. 2019.
- [35] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. *Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts*. 2019. arXiv: 1907.03890 [cs.SE].
- [36] Aina Niemetz and Mathias Preiner. “Bitwuzla”. In: *International Conference on Computer Aided Verification*. Springer. 2023, pages 3–17.
- [37] Louis Noizet and Alan Schmitt. “Necro ML: Generating OCaml Interpreters”. In: (2022).
- [38] Corina S. Păsăreanu and Neha Rungta. “Symbolic PathFinder: Symbolic Execution of Java Bytecode”. In: *IEEE/ACM International Conference on Automated Software Engineering*. 2010.
- [39] *WebAssembly Core Specification*. Version 1.0. W3C, Dec. 5, 2019. URL: <https://www.w3.org/TR/wasm-core-1/>.
- [40] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. “Symbolic Execution for JavaScript”. In: *International Symposium on Principles and Practice of Declarative Programming*. 2018.
- [41] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. “JaVerT 2.0: compositional symbolic execution for JavaScript”. In: *Proceedings of the ACM on Programming Languages* (2019).

- [42] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. “A Symbolic Execution Framework for JavaScript”. In: *IEEE Symposium on Security and Privacy*. 2010.
- [43] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C”. In: *ACM SIGSOFT Software Engineering Notes* (2005).
- [44] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. “MultiSE: Multipath Symbolic Execution Using Value Summaries”. In: *Joint Meeting on Foundations of Software Engineering*. 2015.
- [45] Colibri2 Team. *Colibri2*. URL: <https://colibri.frama-c.com/>.
- [46] The Coq Development Team. *The Coq Reference Manual – Release 8.19.0*. <https://coq.inria.fr/doc/V8.19.0/refman>. 2024.
- [47] Emina Torlak and Rastislav Bodik. “A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014.
- [48] Can I Use. *WebAssembly Browser Support*. Accessed March-2024. URL: <https://caniuse.com/wasm>.
- [49] Dong Wang, Bo Jiang, and W. K. Chan. *WANA: Symbolic Execution of Wasm Bytecode for Cross-Platform Smart Contract Vulnerability Detection*. 2020. arXiv: 2007.15510 [cs.SE].




## About the authors


**Léo André** l@ndrs.fr

 <https://orcid.org/0000-0003-2940-6605>


**Filipe Marques** filipe.s.marques@tecnico.ulisboa.pt

 <https://orcid.org/0000-0002-2555-5382>


**Pierre Chambart** pierre.chambart@ocamlpro.com

 <https://orcid.org/0009-0008-9163-9091>

**Arthur Carcano** arthur.carcano@ocamlpro.com

 <https://orcid.org/0000-0002-9946-1645>

**José Fragoso Santos** jose.fragoso@tecnico.lisboa.pt

 <https://orcid.org/0000-0001-5077-300X>