Modular Abstract Definitional Interpreters for WebAssembly

Katharina Brandl
JGU Mainz, Germany

Sebastian Erdweg
JGU Mainz, Germany

Sven Keidel
TU Darmstadt, Germany

Nils Hansen
JGU Mainz, Germany

Abstract

Even though static analyses can improve performance and secure programs against vulnerabilities, no static whole-program analyses exist for WebAssembly (Wasm) to date. Part of the reason is that Wasm has many complex language concerns, and it is not obvious how to adopt existing analysis frameworks for these features. This paper explores how *abstract definitional interpretation* can be used to develop sophisticated analyses for Wasm and other complex languages efficiently. In particular, we show that the semantics of Wasm can be decomposed into 19 language-independent components that abstract different aspects of Wasm. We have written a highly configurable definitional interpreter for full Wasm 1.0 in 1628 LOC against these components. Analysis developers can instantiate this interpreter with different value and effect abstractions to obtain abstract definitional interpreters that compute inter-procedural control and data-flow information. This way, we develop the first whole-program dead code, constant propagation, and taint analyses for Wasm, each in less than 210 LOC. We evaluate our analyses on 1458 Wasm binaries collected by others in the wild. Our implementation is based on a novel framework for definitional abstract interpretation in Scala that eliminates scalability issues of prior work.

2012 ACM Subject Classification Software and its engineering → Automated static analysis

Keywords and phrases Static Analysis, WebAssembly

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

WebAssembly (Wasm) is a low-level programming language targeted at efficient and portable computation on the web [9]. Wasm modules are often used as a drop-in replacement for computation-intensive JavaScript libraries such as game engines [22, 9]. Wasm has also been designed with security in mind, but many security vulnerabilities reemerge in Wasm because OS-level routines must be provided as user code, which makes them susceptible to attacks [19], and because current compilers targeting Wasm lack protection mechanisms such as stack canaries [28]. While it is well-known that static program analyses can drive performance optimization, reduce binary size, and discover vulnerabilities, no static whole-program analyses exist for Wasm to date.

Wasm involves many complex and interacting language features that analyses have to model: operand stacks, call frames, jumps to scoped labels, function and global-variable tables, dynamically loaded modules, and module-owned linear memory to name a few. It is not obvious how to adopt existing analysis frameworks for these features, and it is not obvious how to develop a new analysis framework for these features without years of engineering. But, as this paper will show, *abstract definitional interpretation* comes to the rescue.

Abstract definitional interpretation was first proposed by Darais et al. [7] as an alternative to abstracting abstract machines [11]. The key idea is to define a generic definitional interpreter that is parametric in value and effect operations, such that it can be instantiated to form concrete as well as abstract interpreters. Keidel et. al. [13] refined this approach to isolate and permit modular reasoning about value and effect components [12]. However, abstract definitional interpretation has only been used for small languages (PCF, subsets of Scheme) and toy programs so far. It is unclear if abstract definitional interpretation scales to languages as complex as Wasm and if the resulting analyzers scale to real-world programs of considerable size. In this paper, we answer both of these questions affirmatively and explain how we developed three Wasm analyses in less than 210 LOC each.

The foundation of all our Wasm analyses is a generic definitional interpreter for Wasm, which we designed and implemented. An important contribution of this paper is to decompose the semantics of Wasm and map it to 12 value components and 7 effect components. Each component consists of an interface with a canonical concrete semantics and any number of abstract semantics. Since these components are language-independent, we only have to develop them once and can reuse them across languages and analyses. This way, we managed to develop a fully-fledged definitional interpreter for Wasm 1.0 and its module system in only 1628 lines of language-dependent code.

The generic interpreter is implemented against the interfaces of value and effect components, making the mapping from language concerns to components explicit. Analysis developers can derive abstract definitional Wasm interpreters by selecting an implementation for each component used by the generic interpreter. This makes analysis development modular: We can reuse components between analyses and refine individual components while reusing others unchanged.

We instantiate the generic interpreter modularly to derive three abstract definitional interpreters for Wasm: a context-insensitive dead code analysis based on an inter-procedural control-flow graph that we compute, a callsite-sensitive constant propagation analysis, and a callsite-sensitive taint analysis. Each of these analyses is novel for Wasm, and each of them required less than 210 lines of code shown in the table to the right.

Component	Lines of Code
Generic Interpreter	1628
Dead Code Analysis	130
Constant Analysis	156
Taint Analysis	209

Technically, our implementation is based on a new framework for definitional abstract interpretation in Scala. Our framework improves over the original DAI by Darais et al. [7] and Sturdy by Keidel et al. [12] to make definitional abstract interpreters scalable. Specifically, our framework exploits a simpler component design and eliminates the monadic transformer stack required by DAI and Sturdy. We show that our analyses scale to real-world programs by analyzing 1458 Wasm binaries collected by others in the wild. Since these binaries are not full applications, we also developed a most general client for Wasm that allows us to apply our whole-program analyses to individual modules soundly. On average, each of our analyses takes 5s per binary, and we find 14% of all instructions are dead code, 10% of all instructions could be replaced by constants, and 56% of all memory accesses are safe against tampering.

In summary, we make the following contributions:

- We present the design of a modular analysis platform for Wasm (Section 3).
- We decompose Wasm into 12 value components and 7 effect components and implement a generic interpreter against their interfaces (Section 4).

	Concrete	Concrete	Type Abs.
(func (param i64)	param=1	param=4	param=i64
(result i64)	result=1	result=24	result=i64
local i64			
i64.const 1	[1]	[1]	[i64]
local.set 1	[]	[]	[]
(loop	[]	[]	[]
local.get 0	[1]	[4]	[i64]
i64.const 1	[1,1]	[1,4]	[i64,i64]
i64.le_u	[1]	[0]	[i32]
(if	[]	[]	[]
(then	[]		[]
local.get 1	[1]		[i64]
return)	[]		[]
(else		[]	[]
local.get 1		[1]	[i64]
local.get 0		[4,1]	[i64,i64]
i64.mul		[4]	[i64]
local.set 1		[]	[]
local.get 0		[4]	[i64]
i64.const 1		[1,4]	[i64,i64]
i64.sub		[3]	[i64]
local.set 0		[]	[]
br 1))))		...	...

Figure 1 Factorial function in Wasm: Two concrete runs and an abstract run using a type-based domain.

- We modularly define 3 whole-program analyses that are novel for Wasm and provide a most general client for Wasm modules (Section 5).
- We designed and implemented a new, scalable framework for abstract definitional interpreters in Scala and explain how it improves over prior work. We realized our modular analysis platform for Wasm on top of this framework (Section 6).
- We validate the soundness, performance, and usefulness of the Wasm analyses (Section 7).

2 Introduction to WebAssembly and Problem Statement

Wasm is a low-level stack-based programming language with structured control flow. We illustrate the textual syntax and some of the core features of Wasm using an iterative factorial function in Figure 1 as an example. The leftmost column shows the code of the factorial function, whereas the other columns display the stack of the concrete and abstract executions of that code. Note that the local variable at index `0` refers to the function parameter and is used as an iteration counter, whereas the local variable at index `1` is an accumulator for the result of the factorial function.

We illustrate the concrete interpretation of the factorial function for arguments `1` and `4`. Most Wasm operations interact with the operand stack whose contents we show in Figure 1 for each instruction. For example, `i64.const` and `local.get` push values to the stack, whereas `local.set` and `i64.le_u` pop values from the stack. For `param=1`, the `if` finds that the argument is less-equal than `1` and thus terminates. For `param=4`, the `if` goes to the else-branch, where we accumulate the factorial result, decrement the iteration counter, and jump to the beginning of the loop. Jumps in Wasm are structured, which means they can only target enclosing blocks, indexed by distance. In our example, `br 1` jumps over the if-block and targets the loop. After a few more iterations, we will again reach the then-branch where the loop terminates.

To illustrate the abstract interpretation of Wasm, the rightmost column in Figure 1 shows an abstract evaluation of the factorial function where values are approximated by their type. The factorial function is called with type `i64` as argument, denoting any 64-bit integer. Each abstract evaluation must overapproximate both concrete evaluations. Hence the abstract interpreter analyzes both branches of the if-instruction and loop until reaching a fixed point. This type analysis can be used to derive a control-flow graph, but the value representation is configurable in our system. Later in this paper, we present Wasm analyses that use more precise value abstractions.

Wasm provides many other interesting features not shown in our illustrating example. For instance, in addition to normal function calls, there are also indirect function calls whose call target can be found in a function table. Functions can also be imported from other modules and Wasm code can invoke external functions provided by the runtime system. When Wasm runs in the browser, these external functions are JavaScript programs. Finally, each Wasm module can declare module-global variables and request a linear memory (i.e., a byte array) to store data.

Problem Statement. We want to develop abstract interpreters for Wasm that track data-flow and information-flow. This is a difficult challenge since the abstract interpreter has to deal with all of Wasm's concerns: the operand stack, call frames, global variables, linear memory, function tables, and structured jumps. Handling all these concerns at once complicates the implementation of the abstract interpreter and makes it harder to change individual aspects later.

For example, consider the semantics of indirect function calls which combines 5 Wasm concerns highlighted with italic font: The interpreter first pops the *numeric index* of the function from *operand stack* and uses it to search through the *function table* to find the function definition. If the table has a function definition of the correct type at the index, the interpreter invokes the function. In particular, the interpreter binds the function arguments on the operand stack to the function parameters on a newly created *call frame*. Finally, the interpreter processes the body of the function and afterwards pushes the return argument on the stack. There are also multiple edge cases which cause the function invocation to *fail*.

A naive monolithic analysis implementation may closely couple the semantics of indirect calls to specific abstractions for the function index, the operand stack, call frame, and failures. This coupling not only complicates the analysis implementation, it also makes it difficult to change the abstractions without also requiring changes to the abstract semantics of indirect calls. To solve this problem, we divide and conquer by modularizing the analysis implementation, which we discuss in the following section.

3 Modular Wasm Analyses in a Nutshell

In this section, we present the design of our modular analysis platform for Wasm. At the core of our platform is a generic definitional interpreter for Wasm. The generic interpreter describes the semantics of Wasm instructions and serves as a template to derive different Wasm analyses, as well as a concrete interpreter. The generic interpreter is parametric in its representation for values such as integers and floating point values. Furthermore, the generic interpreter is parametric in its representation of effects such as the linear memory or the operand stack. Analyses instantiate the generic interpreter with different abstractions for values such as constants, taint flags, or types and with different abstraction for effects such as a constant memory abstraction. Similarly, the concrete interpreter instantiates the generic interpreter with concrete values and effects.

Our platform is modular along two dimensions. First, the generic interpreter defines

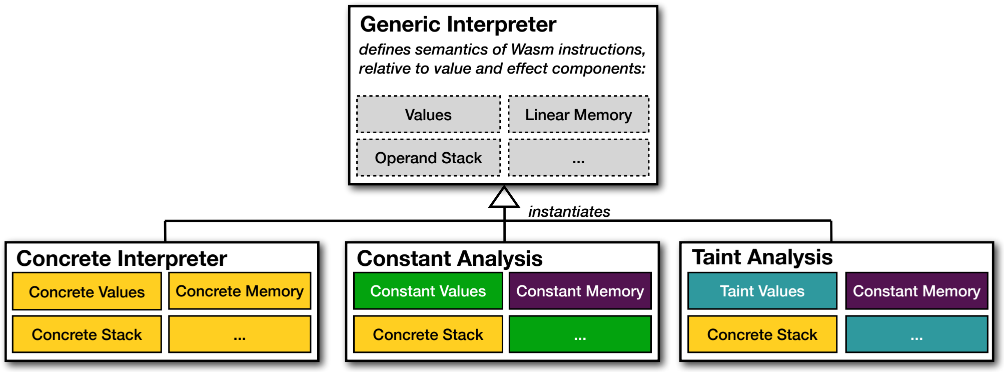


Figure 2 Overview of our modular Wasm analysis platform

```
trait GenericInterpreter[V, ExcV]:
  // Independent value components for abstract value type V
  val i32ops: IntegerOps[Int, V]
  val f64ops: FloatOps[Double, V]
  // Independent effect components
  val stack: OperandStack[V]
  type WasmExc[V] = (JumpTarget, List[V])
  val except: Except[WasmExc[V], ExcV]
  // Interpreter written against value and effect components
  def evalInst(inst: Inst): Unit = inst match
    case i32.Sub =>
      val v2 = stack.popOrFail()
      val v1 = stack.popOrFail()
      stack.push(i32ops.sub(v1,v2))
    case f64.Abs =>
      val v = stack.popOrFail()
      stack.push(f64ops.abs(v))
    case Return =>
      val operands = stack.popNOrFail(currentReturnArity)
      except.throws((JumpTarget.Return, operands))
```

Figure 3 A simplified generic interpreter for Wasm that handles subtraction, absolutes, and function returns.

the semantics for Wasm instructions once and for all; analyses simply reuse that semantics. Second, the values and effects required by the generic interpreter are decomposed into language-independent components, which can be defined language-independently and reused flexibly. Figure 2 illustrates the modularity of our platform. The generic interpreter sits on top and is instantiated to obtain concrete and abstract interpreters. It depends on various value and effect components that must be provided during instantiation. In Figure 2, the colors illustrate component reuse. While each interpreter uses a different value representation, the two abstract interpreters use the same component for linear memory and the operand stack. Since the shape of the operand stack is decidable in Wasm [9], this component is also shared with the concrete interpreter. In the remainder of this section, we illustrate how our analysis platform realizes the generic interpreter, its instances, and the components.

Generic interpreter. Figure 3 shows a simplified generic interpreter for Wasm. The generic interpreter does not refer to any specific concrete or abstract value representations. Instead, the interpreter abstracts over them with the value components `IntegerOps` for 32-bit integers and `FloatOps` for 64-bit floats. Value components are interfaces with any number of implementations, for example:

```
trait IntegerOps[B, V]:       // a type class for integer operations
  def integerLit(i: B): V     //   - embeds base literals of type B into the value type V
  def sub(v1: V, v2: V): V    //   - subtraction of two values
object ConcreteIntegerOps extends IntegerOps[Int, Int] {...}         // concrete semantics
object ConstantIntegerOps extends IntegerOps[Int, Topped[Int]] {...} // constant abstraction
object SignLongIntegerOps extends IntegerOps[Long, Sign] {...}       // sign abstraction
```

In addition to the value components, the simplified generic interpreter requests two components for effects: one for the mutable operand stack and one for exception handling. Like value components, effect components define an interface that can be implemented in various ways. The `OperandStack[V]` effect component provides `push`, `pop`, and `peek` operation for values of type `V`. The `Except` component provides operations for throwing and catching exceptions of type `WasmExc[V]`, consisting of a jump target and a list of operand values. In contrast to prior frameworks for abstract definitional interpretation, we distinguish value from effect components to improve the run-time performance of our analyses. Specifically, value components capture pure operations and do not contribute to the analysis state, whereas effect components maintain internal state that is part of the overall analysis state. This becomes relevant when joining computations or computing the fixpoint of an analysis.

The generic interpreter only relies on the interfaces of value and effect components. Based on these, the generic interpreter defines the semantics of Wasm instructions with the interpretation function `evalInst`. We only show a few selected cases. For integer subtraction, function `evalInst` pops two values from the stack, subtracts them, and pushes the result back on the stack. Note that Wasm instructions are not overloaded, so it is easy to select the appropriate value component. For example, function `evalInst` delegates the instruction `f64.Abs` to the component `f64ops`, which handles 64-bit floating-point numbers. The operand stack is ubiquitous in the generic interpreter, but other effects are needed too. For example, function `evalInst` implements return instructions using exceptions that are caught at the function head. Exception handling is a standard way for implementing non-local control flow on the JVM, where our analyzers run. Exception handling also closely aligns with jumps and returns in Wasm: Due to the structured control flow of Wasm, all jumps (including returns) target a surrounding block. Similarly, exceptions interrupt execution and return to the closest surrounding exception handler.

Concrete interpreter. We can instantiate the generic interpreter for different value and effect components. In particular, we can derive a concrete Wasm interpreter by choosing the canonical concrete semantics for all components and lifting them to Wasm values. Specifically, we represent Wasm values using the corresponding number types of the JVM.

```
enum Value:
  case I32(i: Int); case I64(l: Long); case F32(f: Float); case F64(d: Double)
```

With this, we can instantiate the generic interpreter:

```
class ConcreteInterpreter extends GenericInterpreter[Value, WasmExc[Value]]:
  val i32ops = ... // lifts IntegerOps[Int, Int] to Value
  val f64ops = ... // lifts FloatOps[Double, Double] to Value
  val stack = new ConcreteOperandStack[Value]
  val except = new ConcreteExcept[WasmExc[Value]]
```

For values we lift the canonical concrete semantics to the `Value` type, for effects we select all required effect components directly from our library.

Abstract interpreter. We can derive abstract interpreters in the same manner. For example, let us build a type analysis:

```
enum Type:
  case I32; case I64; case F32; case F64; case Top
```

We instantiate the generic interpreter using `Type` for values and by joining exceptions that jump to the same target:

```
type ExcByTarget = Map[JumpTarget,List[Type]]
class AbstractInterpreter extends GenericInterpreter[Type, ExcByTarget]:
  val i32ops = // lifts IntegerOps[Int, BaseType[Int]]
  val f64ops = // lifts FloatOps[Double, BaseType[Double]]
  val stack = new JoinableConcreteOperandStack[Type]
  val except = new JoinedExcept[WasmExc[Type], ExcByTarget]
```

Our platform provides language-independent type abstractions for various components using a singleton type `BaseType[T]` to represent type `T`. For the value components in Wasm, we select and lift these abstractions from `BaseType` to `Type`. For the operand stack, we exploit that its shape is decidable for Wasm, which allows us to reuse the concrete operand stack (through subclassing). The abstract interpreter must join the contents of stacks at control-flow join points, but these stacks will have equal size. For exceptions, we select an abstract semantics that collects all possibly active exceptions in a set. Although not shown here, analyses can select a context-sensitivity and configure other aspects of the fixpoint algorithm, such as the iteration strategy or loop unrolling depth.

This example illustrates how our platform supports the modular development of Wasm analyses: by plugging together value and effect components and instantiating the generic interpreter. Moreover, individual components can be refined and replaced easily. But how can we decompose Wasm into value and effect components and define a generic interpreter for the full language?

4 Decomposing Language Concerns of WebAssembly

In this section, we propose a decomposition of Wasm that separates individual language concerns into components. We will then define a Wasm generic interpreter on top of these components. The generic interpreter only uses the interfaces of the components, while concrete and abstract interpreters instantiate the generic interpreter with selected implementations of the components. This way, the decomposition of Wasm into components enables analysis developers to compose full-fledged Wasm analyses modularly.

In the remainder of this section, we present our decomposition of Wasm and its mapping to value and effect components. For each component, we have implemented the canonical concrete semantics compatible with the Wasm specification. We show possible abstract semantics in Section 5, where we construct data and information-flow analyses for Wasm.

4.1 Values

Wasm defines four different value types, namely integers and floats with 32 and 64 bits: `i32`, `i64`, `f32`, `f64`. In Section 3, we already showed how some of the value components can be used to implement value operations generically, such as `IntergerOps` for implementing operations on integers. However, we omitted many details for illustration purpose. The goal of this subsection is to fill the gap and to introduce other value components we used for Wasm. Throughout this section, the type variable `V` stands for the abstract value type used by the generic interpreter.

Numeric operations We decompose the numeric operations of Wasm into 6 value components. Besides components for the various arithmetic operations of the four value types, we use one component for equality testing, and one component for ordering comparisons of Wasm values:

```
val i32ops: IntegerOps[Int, V]
val i64ops: IntegerOps[Long, V]
val eqOps: EqOps[V, V]
```

```
val f32ops: FloatOps[Float, V]
val f64ops: FloatOps[Double, V]
val orderingOps: OrderingOps[V, V]
```

The mapping from Wasm instructions to the respective components is straightforward, although some instructions are implemented on top of multiple component operations, such as:

```
def evalIUnop(op: IUnop, v: V): V = op match
  case i64.Extend32S =>
    val shift = i64ops.integerLit(32)
    i64ops.shiftRight(i64ops.shiftLeft(v, shift), shift)
```

Also note that the validation of Wasm rejects comparisons on values of different type. Thus, when providing instances for `EqOps` and `OrderingOps`, it is sufficient to consider those cases where the operands have the same type.

Conversions. Wasm features many operations that convert between value types. For example, there are three operations converting from `i32` values to `f32` values, namely signed and unsigned conversions and byte reinterpretation. We use a single `Convert` interface for all conversions, but require 12 different instances of that component:

```
trait Convert[From, To, VFrom, VTo, Config]:
  def apply(from: VFrom, conf: Config): VTo

val convert_i32_i64: Convert[Int, Long, V, V, ..]
val convert_i32_f32: Convert[Int, Float, V, V, ..]
val convert_i32_f64: Convert[Int, Double, V, V, ..]
...
```

Note that the first two type parameters `From` and `To` of `Convert` are tags or phantom types: They are only used to describe the component. The actual values to be converted are of type `VFrom` and `VTo`, both of which we instantiate with `V` in the generic interpreter. Actual instances consider specific value representations for `VFrom` and `VTo`, and we lift these instances to operate on values `V` as described below. The `Config` parameter guides the conversion. For example, the following code handles the three different conversions of `i32` to `f32` values:

```
def evalConvertop(op: Convertop, v: V): V = op match
  case f32.ConvertSI32 => convert_i32_f32(v, Signed)
  case f32.ConvertUI32 => convert_i32_f32(v, Unsigned)
  case f32.ReinterpretI32 => convert_i32_f32(v, Raw)
```

The `Convert` interface can not only be used for numeric conversion operations. We use the same interface for operations that serialize and deserialize values into bytes. This is required to write values into Wasm's linear byte memory:

```
val encode: Convert[V, Seq[Byte], V, Bytes, ...]
val decode: Convert[Seq[Byte], V, Bytes, V, ...]

def evalInst(inst: Inst): Unit = inst match
  case i: StoreInst =>
    val v = stack.popOrFail()
    val bytes = encode(v, ...)
    ... // store bytes in memory
```

Branching. Concrete and abstract interpreters differ significantly when it comes to branching control flow, as required for conditional constructs. While the concrete interpreter will select exactly one branch to execute, abstract interpreters must analyze both branches unless they can statically decide if the branching condition is true or false. We capture branching with a value component that receives two continuations:

```
trait BoolBranching[B, R]:
  def boolBranch(v: B, thn: => R, els: => R): R
```

Implementations of this interface can select the type `B`, for which they can decide the branching. For example, we show the canonical concrete semantics using `Boolean` for `B` and a type semantics using `BaseType[Boolean]`:

```
class ConcreteBranch[R] extends BoolBranching[Boolean, R]:
  def boolBranch(v: Boolean, thn: => R, els: => R): R = if (v) thn else els

class BaseTypeBranch[R](effects:EffectStack,j:Join[R]) extends BoolBranching[BaseType[
     Boolean],R]:
  def boolBranch(v:BaseType[Boolean], thn:=>R, els:=>R): R = effects.joinComputations(thn,
     els, j)
```

The concrete semantics simply uses the Boolean condition to decide which branch to execute. In contrast, the type semantics must execute both branches and join their results and effects. Our platform provides a helper function `joinComputations` to achieve that, given the stack of effects (`EffectStack`) used by the abstract interpreter and an instance of type class `Join[R]`. In our implementation, these arguments are modeled as implicit parameters and resolved automatically. We explain how our framework joins effectful computations in more detail in Section 6.

The generic interpreter uses `boolBranch` for all conditional Wasm instructions: `select`, `brif`, and `if`. For example:

```
val branchOps: BooleanBranching[V, Unit]

def evalInst(inst: Inst): Unit = inst match
  case If(bt, thnInsts, elsInsts) =>
    val isZero = evalInst(i32.Eqz)
    branchOps.boolBranch(isZero,
      label(elsInsts), label(thnInsts))
```

We will explain the `label` function later in the context of jumps. For now it is sufficient to know that it executes a labeled block of code.

Lifting Value Components. Our platform provides language-independent concrete and abstract instances for all value components, such as the concrete `IntergerOps[Int, Int]` and the abstract `IntegerOps[Int, BaseType[Int]]`. However, as shown above, generic interpreters usually require operations on some compound type for values. To reuse the language-independent component instances, we must lift them to the Wasm-specific value type. To facilitate this, our platform provides lifting instances for all value components, which can be easily instantiated. For example, the following two definitions lift the concrete and type-based integer operations to Wasm values and types, respectively:

```
val i32opsValue: IntegerOps[Int, Value] =
  new LiftIntegerOps({case Value.I32(i) => i}, i => Value.I32(i))
```

```
val i32opsType: IntegerOps[Int, Type] =
  new LiftIntegerOps({case Type.I32 => BaseType[Int]}, _ => Type.I32)
```

For an underlying value type `U`, `LiftIntegerOps` take an extract function `V => U` and an inject function `U => V`. With these, it wraps the operations of the underlying language-independent component instance, for example:

```
def sub(v1: V, v2: V): V = inject(underlying.sub(extract(v1), extract(v2)))
```

In our Wasm analyses, all value components are based on language-independent component instances that we lift.

4.2 Effects

Computations generally yield values and trigger effects. Wasm features many language concerns that are effectful. We capture these concerns in effect components. While value components are stateless, effect components contain internal state. This distinction is important when joining computations (as in the type-based `boolBranch`), because effect components must participate in the join (see Section 6 for details). In this subsection, we present a decomposition of Wasm's effectful language concerns into effect components.

Operand Stack Wasm programs interact with an operand stack. We capture this effect in a dedicated effect component:

```
trait OperandStack[V, MayJoin[_]]:
  def push(v: V): Unit
  def pop(): JOption[MayJoin, V]
  def popOrFail(): V = ...
  ...
```

Except for the `MayJoin` type parameter, this component provides a standard stack interface. The `MayJoin` parameter determines whether the component can yield an uncertain result for `pop`. For example, if an abstract stack semantics lost track of the stack's height, `pop` would yield an uncertain result that comprises alternative values or even a stack underflow. In contrast, a concrete stack semantics yields certain results only: either the stack's topmost value or no value if the stack is empty. Instances of `OperandStack` can declare which behavior they provide by choosing `NoJoin` or `WithJoin` for `MayJoin`:

```
enum MayJoin[A]:
  case NoJoin()
  case WithJoin(j: Join[A], eff: EffectStack)
```

Indeed, a concrete stack uses `NoJoin` whereas an abstract stack uses `WithJoin`. Given a `WithJoin[A]`, we can invoke `joinComputations` as shown above in the abstract branching semantics of Section 4.1. `OperandStack` forwards the `MayJoin` parameter to `JOption`, a data type for joinable option values that we use to represent uncertain data. Since `JOption[NoJoin, A]` is isomorphic to the standard `Option[A]`, concrete operand stacks provide a standard stack interface.

Many of our effect components use a similar design to declare that operations may yield uncertain results in the abstract semantics. Indeed, the generic interpreter itself has a `MayJoin` parameter that it forwards to the required effect components. However, sometimes the generic interpreter can formulate more precise requirements. For Wasm, the language specification guarantees that the height of the stack is decidable at all times and that stack lookups must yield certain results. To this end, the generic Wasm interpreter requires a decidable operand stack, which internally selects `NoJoin` for `MayJoin`:

```
val stack: DecidableOperandStack[V]
```

Indirect Calls and Function Tables Wasm features indirect function calls via function indices, which really are plain `i32` values computed by the program. To evaluate an indirect function call, Wasm reads a function index from the stack, looks up the index in a function table, and invokes the found function:

```
def evalInst(inst: Inst): Unit = inst match
  case CallIndirect(typeIx) =>
    val funcIx = stack.popOrFail()
    val funV = funTable.getOrElse(funcIx, fail(...))
    funOps.invokeFun(funV, invoke)
```

This code uses two additional components: an effect component `funTable` and a value component `funOps`. We model the function table as a generic `SymbolTable` component that maps symbols to entries:

```
trait SymbolTable[Symbol, V, MayJoin[_]]:
  def get(symbol: Symbol): JOption[MayJoin, V]
  def put(symbol: Symbol, newEntry: V): Unit

val funTable: SymbolTable[FuncIx, FunV, MayJoin]
```

Note how the symbol table uses the same `MayJoin` pattern as the operand stack. However, lookups in the function table are not decidable in Wasm, so that abstract interpreters sometimes obtain an uncertain function. For example, our type analysis does not track the values of function indices and thus must consider all reachable functions as potential targets for indirect calls. This also is the reason why the function table contains `FunV` values rather than functions directly: We must be able to join function values. To abstract from the specific `FunV` representation, we use a generic value component `FunctionOps`:

```
trait FunctionOps[Fun, A, R, FunV]:
  def funValue(fun: Fun): FunV
  def invokeFun(v: FunV, a: A)(invoke: (Fun, A) => R): R

val funOps: FunctionOps[Function, FuncType, Unit, FunV]
```

Operation `funValue` lifts a function into a function value `FunV`. Operation `invokeFun` does the inverse: It extracts functions from a function value and applies the continuation `invoke` on each of them. Similar to `boolBranch`, abstract instances of `FunctionOps` join the result `R` of all functions.

Global Variables Wasm features numerically indexed global variables that can be used to store values. We model global variables using the same `SymbolTable` component that we used for function tables. However, the resolution of global variables is decidable in Wasm and always yields a certain result. We incorporate this fact in the generic interpreter by requiring a decidable symbol table for global variables:

```
val globals: DecidableSymbolTable[Int,V]
```

Please note that in Wasm, each module has its own globals, function table, and memory, which can also be shared between modules. Our implementation takes this into account, but we decided to simplify the presentation of the code for the paper.

Local Variables Each Wasm function can declare local variables, which we understand to include the function parameters. A function can read and write its local variables freely. We model local variables through a generic `CallFrame` component. Each call frame has a fixed size determined at construction by operation `inNewFrame`. In addition, a call frame can track auxiliary `Data` for each frame. For Wasm, we use the call frame to track the module instance of the currently executing function as well as its return arity:

```
trait CallFrame[Data, Var, V, MayJoin[_]]:
  def inNewFrame[A](d: Data, vs: Seq[(Var, V)])(f: => A): A
  def getFrameData: Data
  def getLocal(x: Int): JOption[MayJoin, V]
  def setLocal(x: Int, v: V): JOption[MayJoin, Unit]

val callFrame: DecidableCallFrame[(ModuleInst, Int), Int, V]
```

Note how both call frames and symbol tables map indices to values. However, call frames are scoped by function call and the previous call frame is restored when exiting a function. Operation `inNewFrame` takes care of this behavior, executing `f` in the new frame and restoring the previous frame after `f` finishes. This way, the generic interpreter can implement function invocations:

```
def invoke(fun: Function): Unit =
  val args = stack.popNOrFail(fun.params.size)
  val locals = args ++ fun.locals.map(num.defaultValue)
  val data = (module, fun.returnArity)
  callFrame.inNewFrame(data, locals)(enterFunction(fun))
```

Linear Memory Wasm programs can load and store data from a growable linear memory. Technically, the linear memory is a byte array that is accessed using 32-bit integers as index. Wasm provides various instructions to load and store values of different types. In our generic interpreter, the following code handles load instructions using the memory effect component:

```
trait Memory[Addr, Bytes, Size, MayJoin[_]]:
  def read(addr: Addr, length: Int): JOption[MayJoin, Bytes]
  def write(addr:Addr, bytes:Bytes): JOption[MayJoin, Unit]

val memory: Memory[Addr, Bytes, Size, MayJoin]
def load(inst: LoadInst): Unit =
  val addr = effectiveAddr(inst.offset)
  val length = getBytesToRead(inst)
  val bytes = memory.readOrElse(addr, length, fail(...))
  stack.push(decode(bytes, inst))
```

We first compute the effective address to be loaded by adding a static offset to the base address, which is on the operand stack. We then determine the number bytes to be loaded. We invoke the read operation of the memory effect component to obtain a byte sequence. Finally, we decode those bytes using the `decode` component discussed in Section 4.1.

Jumps Wasm features a limited form of jumps that abides by structured control flow, which means that jumps can only target enclosing blocks. Instead of using named labels, Wasm jumps declare the number of blocks to skip, that is, the block-distance between the jump and the target block. We model jumps through a effect component for exception handling:

```
trait Except[Exc, ExcV, MayJoin[_]]:
  def throws(ex: Exc): Nothing
  def tries[A](f: => A): JEither[MayJoin, A, ExcV]
```

The `Except` component is parametric in the underlying exception type `Exc` and the representation of exception values `ExcV`. Similar to `JOption` from above, operation `tries` yields a value of a joinable either data type, `JEither` for short. That is, `tries` either yields an `A` when `f` triggers no exception, or it yields an `ExcV`. Since abstract instances of `Except` may not be able to determine the exact behavior of `f`, the result of `tries` can be uncertain, which `JEither` encapsulates.

The generic interpreter uses exception handling to support jumps and returns:

```
type WasmExc[V] = (JumpTarget, List[V])
enum JumpTarget:
  case Jump(labelIndex: LabelIdx)
  case Return

val except: Except[WasmExc[V], ExcV, MayJoin]

def jump(labelIndex: LabelIdx): Unit =
  val returnArity: Int = labelStack.arityOf(labelIndex)
  val operands = stack.popNOrFail(returnArity)
  except.throws((JumpTarget.Jump(labelIndex), operands))

def label(returnArity: Int, insts: Seq[Inst]): Unit =
  labelStack.pushLabel(returnArity)
  val tried = except.tries(insts.foreach(evalInst))
  labelStack.popLabel()
  tried.either(identity) {
    case (JumpTarget.Jump(0), ops) => stack.pushN(ops)
    case (JumpTarget.Jump(ix), ops) => except.throws(WasmExc.Jump(ix - 1, ops))
    case (JumpTarget.Return, ops) => except.throws(WasmExc.Return(ops))
  }
```

Function `jump` takes the index of a label, looks up the return arity required by that label in an auxiliary data structure called `labelStack`, and triggers a `Jump` exception with the corresponding number of operands. Jump exceptions are handled by function `label`, which we use when entering a new block. This function first pushes the return arity of the label to the `labelStack` and then tries to run all instructions of the block. We use `either` to react to the result of that execution. If the block succeeds without exception, nothing has to be done (`identity`). However, if an exception was (possibly) thrown, we react accordingly. If the jump target has index `0`, it targets the current label and we push the operands on the stack. Otherwise, we decrement the jump target index and escalate the exception. Return exceptions always escalate; they are handled by `enterFunction`.

Traps. Wasm programs can trigger unrecoverable errors, called traps. We model traps using the `Failure` effect.

```
trait Failure:
  def fail(kind: FailureKind, msg: String): Nothing

val failure: Failure
```

In contrast to exceptions, failures are unrecoverable and cannot be caught. While the canonical concrete semantics of `Failure` aborts the execution of a Wasm program, abstract interpreters must continue to explore execution paths that do not fail. That is, an abstract interpreter usually produces a set of potential `FailureKind` as part of the final analysis result.

4.3 Summary

We have decomposed Wasm into various language concerns and mapped them to value and effect components. Based on this decomposition, we have developed a generic interpreter for

Wasm that is parametric in how the value and effect components are instantiated. The generic interpreter implements evaluation of Wasm code. The generic interpreter also implements the module system, manages exports, resolves imports, and performs module instantiation, which is used to initialize variables, function tables, and memories. In particular, we have implemented the canonical concrete semantics for all value and effect components and used those to derive a concrete Wasm interpreter. This concrete Wasm interpreter is a feature-complete and correct implementation of the Wasm 1.0 specification, as we detail in Section 7.

The generic interpreter is not only parametric in the value and effect components, but also in the fixpoint algorithm. While the concrete interpreter can simply run a program until it terminates, abstract interpreters must widen analysis results to ensure termination. To this end, our generic interpreter is written in an open recursive style, giving control to the fixpoint algorithm in each recursive invocation. When instantiating the generic interpreter, we configure a generic fixpoint algorithm provided by our platform to select context-sensitivity and other aspects. We illustrate such configuration in the next section, where be build three whole-program Wasm analyses as instances of the generic interpreter.

5 Modularly Defined Analyses for Wasm

In the previous section, we have presented the key ingredients of our modular static analysis platform for Wasm: a Wasm semantics decomposed into value and effect components and a generic Wasm interpreter. In the present section, we demonstrate how our platform can be used to implement Wasm analyses modularly. To this end, we implement three Wasm analyses: a dead code analysis, a constant propagation analysis, and a taint analysis. We compose each analysis modularly from value and effect components that we use to instantiate the generic interpreter.

5.1 Dead Code Analysis

The first analysis we want to build is a dead code analysis. To this end, we must construct an inter-procedural control-flow graph (CFG) that allows us to identify unreachable instructions. Note that the construction of a precise CFG is undecidable in general and approximation is required. In this subsection, we use a type analysis to approximate the behavior of the program.

Our platform provides a reusable singleton type `BaseType[T]` to represent type `T`, which we use to model our type analysis:

```
enum Type:
  case I32(i: BaseType[Int]);   case I64(l: BaseType[Long]);
  case F32(f: BaseType[Float]); case F64(d: BaseType[Double]); case Top

type Addr = BaseType[Int]                    type FuncIx = BaseType[Int]
type Bytes = BaseType[Seq[Byte]]             type FunV = Powerset[FunctionInstance]
type Size = BaseType[Int]                    type ExcV = Map[JumpTarget,List[Type]]
```

The type analysis does not track memory access precisely: all reads yield a top value. Specifically, we represent addresses `Addr`, byte sequences `Bytes`, and memory size `Size` using their type. We also don't track function indices: Indirect function calls resolve to the set of all functions currently in the function table. For exceptions, we collect all active exceptions in a set. Based on these definitions, we select the following effect components:

```
val stack = new JoinableConcreteOperandStack[Type]
val memory = new TopMemory[MemoryAddr, Addr, Bytes, Size]
val globals = new JoinableConcreteSymbolTable[GlobalAddr, Type]
val funTable = new UpperBoundSymbolTable[TableAddr, FuncIx, FunV]
val callFrame = new JoinableConcreteCallFrame[FrameData, Int, Type]
val except = new JoinedExcept[WasmException[Value], ExcV]
val failure = new AFailureCollect
```

Note how we use decidable instances for the operand stack, call frames, and global variables, since all three concerns are statically decidable in Wasm. The memory yields top on every read, the function table yields all stored entries when queried. We use the `AFailureCollect` instance for abstract failures, which collects all possible failures of the analyzed program.

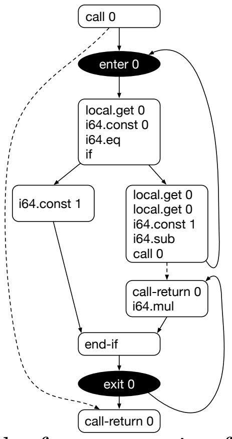
Finally, every analysis must configure the fixpoint algorithm used by our platform. Most importantly, we must select a context-sensitivity and iteration strategy. Our platform provides a combinator library for describing these aspects:

```
val phi = fix.contextSensitive(fix.context.none, fix.filter(isFunOrLoop,fix.iter.innermost))
```

This configuration determines that the type analysis is context-insensitive and uses the `innermost` iteration strategy on functions and loops, meaning nested loops are fixed first.

CFG construction. Our platform uses big-step abstract interpretation, in which the control flow of analyzed programs is implicit. However, we can make the control flow explicit by observing the order in which instructions are executed by the abstract interpreter. To this end, our platform provides generic facilities for observing the control flow and constructing inter-procedural CFGs. For Wasm, we only need to map Wasm constructs to CFG nodes:

```
val cfg = fix.control(config) {
  // called before interpreting an instruction
  case Enter(fun) => CfgEnter(fun)
  case Eval(c: Call, loc) => CfgCall(c, loc)
  case Eval(inst, loc) => CfgInstruction(inst, loc)
} {
  // called after interpreting an instruction
  case (Enter(fun), Exit(_)) => CfgExit(fun)
  case (Eval(c: Call,loc), _) => CfgCallReturn(c, loc)
}
analysis.addContextSensitiveObserver(cfg.logger)
```



For example, this code constructs the CFG shown on the right for a recursive factorial function, where dashed lines represent call-return edges. Of course, the CFG construction also scales to larger examples. The last line in the code above activates CFG logging for a given `analysis`. While our type analysis is context-insensitive, other analyses may exploit context-sensitive CFGs. But, as we show in Section 7, even the simple type analysis already produces useful results and finds dead code in Wasm programs. Furthermore, the CFG can be used as a starting point for other analysis approaches.

5.2 Constant Propagation Analysis

We define a constant propagation analysis by refining the type analysis from above. In a constant propagation analysis, values are either a concrete value or `Top`:

```
enum Value:
  case I32(i: Topped[Int]);   case I64(l: Topped[Long]);
  case F32(f: Topped[Float]); case F64(d: Topped[Double]); case Top
```

```
type Addr = Topped[Int]                      type FuncIx = Topped[Int]
type Bytes = Seq[Topped[Byte]]               type FunV = Powerset[FunctionInstance]
type Size = Topped[Int]                      type ExcV = Map[JumpTarget,List[Type]]
```

Notably, the constant propagation analysis tracks constant memory addresses and bytes. That is, when writing a concrete value to a known address, we store the concrete byte encoding of the value. Conversely, when reading from a known address, if we find a concrete byte sequence, we decode it into a concrete value. This memory abstraction is certainly only a first step in developing sophisticated Wasm analyses, but our modular analysis platform allows us to refine it in future work. For function indices, we track their precise index if possible. Ideally, dereferencing a function index yields a single function that we can execute, but if the function index is `Top`, we obtain a set of all functions in the function table.

Compared to the type analysis, we only have to adapt two effect components, namely those that handle memory and function indices. We highlight the differences in blue font:

```
val stack = new JoinableConcreteOperandStack[Type]
val memory = new ConstantAddressMemory[MemoryAddr, Addr, Bytes, Size]
val globals = new JoinableConcreteSymbolTable[GlobalAddr, Type]
val funTable = new ConstantSymbolTable[TableAddr, FuncIx, FunV]
val callFrame = new JoinableConcreteCallFrame[FrameData, Int, Type]
val except = new JoinedExcept[WasmException[Value], ExcV]
val failure = new AFailureCollect
```

To increase the precision of the constant propagation analysis, we can choose a 1-callsite sensitive fixpoint algorithm. To this end, we log each function call with a call-site logger and use the most recent call site as a context:

```
val callSites = fix.context.callSites {
  case Eval(c: (Call | CallIndirect), _) => Some(c)
  case _ => None
}
val phi = fix.log(callSites,
            fix.contextSensitive(callSites.callString(1),
              fix.filter(isFunOrLoop, fix.iter.innermost)))
```

Finally, we need to determine whether an instruction is constant in all execution paths. We can achieve this by observing the results of the abstract interpreter for each instruction. To this end, we implemented an observer that reads the relevant data from the operand stack before and after executing an instruction. In case an instruction is visited more than once (e.g., in a loop) the recorded values are joined. If the final result is constant, the instruction is constant across all execution paths. Our analysis platform allows us to add this functionality modularly:

```
val constants = new InstructionLogger { inst =>
  // log before execution of inst
  if (readsSingleValueFromStack(inst))
    Some(stack.peekOrFail())
  else if ...
} { inst =>
  // log after execution of inst
  if (writesSingleValueToStack(inst))
    Some(stack.peekOrFail())
  else if ...
}
analysis.addContextFreeObserver(constants)
```

5.3 Taint Analysis

As a last example, we define a taint analysis by refining the constant propagation analysis The goal of the analysis is to detect tainted memory accesses, i.e., if a tainted value is used as memory address. As source for tainted values, we consider input from users and results from calling host function. To track taint, we tag a taint property to each value:

```
enum Value:
  case I32(i: Taint[Topped[Int]]);   case I64(l: Taint[Topped[Long]]);
  case F32(f: Taint[Topped[Float]]); case F64(d: Taint[Topped[Double]]); case Top

type Addr = Topped[Int]                    type FuncIx = Topped[Int]
type Bytes = Seq[Taint[Topped[Byte]]]      type FunV = Topped[Powerset[FunctionInstance]]
type Size = Topped[Int]                    type ExcV = Map[JumpTarget,List[Type]]
```

We omit the effect and fixpoint configuration of the taint analysis since it is identical to the constant propagation analysis.

To detect illegal memory access through tainted values, we add a new observer to the analysis. Note that we observe the values on the stack before they are cast to an address, which is why type `Addr` does not need a taint flag.

```
val tainting = new InstructionLogger { inst =>
  if (isLoadInst(inst)) {
    val addrV = stack.peekOrFail()
    if (addrV.isTainted) Some(Powerset(addrV)) else None
  }
}
analysis.addContextFreeObserver(tainting)
```

We collect tainted addresses for each memory instruction. A memory instruction is safe if its set of tainted addresses is empty. Of course, we could track other sinks or sources for tainted values and expect to do so in future work.

5.4 Most General Client for Wasm Modules

Abstract definitional interpreters are whole-program analyses by construction: Interpretation start in the main function of an application and subsequently explores all code reachable from there. However, Wasm programs are not usually applications, but libraries to be used in JavaScript applications. To apply our whole-program analyses to individual Wasm modules, we develop a most general client for Wasm modules.

Most general clients are often used to apply whole-program static analyses to library code [18]. A most general client approximates all valid usages of a given library, and it can be used as a single entry point for the analysis. We have developed a most general client for Wasm modules that exercises all interleavings of all exported functions in a loop:

```
def runMostGeneralClientLoop(modInst: ModuleInstance)): Unit = {
  effectStack.mapJoin(modInst.exportedFunctions) { case (funcName, funcIx) =>
    val fun = modInst.functions.getOrElse(funcIx, fail(UnboundFunctionIndex, funcIx.toString
     ))
    val args = fun.funcType.params.map(typedTop).toList
    invokeExported(modInst, funcName, args)
  }
  fixpoint(runMostGeneralClientLoop(modInst))
}
```

In each loop iteration, we run all exported functions in isolation and join their effects to update the analysis state. Our fixpoint algorithm iterates this loop until the analysis state is stable. The final analysis state soundly approximates all possible sequences of exported functions, and thus yields a sound analysis result for the library code.

6 A Scalable Framework for Abstract Definitional Interpretation

We designed and implemented a new framework for abstract definitional interpretation in Scala as open source.[1] In this section, we describe how our new framework improves over prior work and why that was necessary for scaling the approach to complex languages and real-world programs. There are two prior frameworks for abstract definitional interpretation: the original DAI in Racket by Darais et al. [7] and Sturdy in Haskell by Keidel et al. [12]. While we compare to both, we also implemented a complete generic definitional interpreter for Wasm in Sturdy and report on the lessons learned.

Component design. Abstract definitional interpretation has supported modularly defined components from the start. Already in DAI, the generic PCF interpreter used components for environments, stores, and allocation [7]. However, these components followed an ad-hoc design and did not share an interface between concrete and abstract semantics. Not only did this preclude modular reasoning about components, it also implies that we must use the non-determinism monad to collect alternative analysis (sub-)results. For example, DAI features a function `isZero(v: V): Boolean` in the concrete semantics and `isZero(v: V): List[Boolean]` in the abstract semantics. Consequently, when the abstract semantics cannot decide if a value is zero it yields `List(true, false)` and all of the remaining analysis is run twice: once for `true` and once for `false`. Nested conditionals with uncertain conditions like this trigger an exponential blow-up that is unacceptable when scaling up.

Sturdy was designed to support the development of sound static analyses with compositional soundness proofs. For this reason, Sturdy introduced a design principle based on parametricity that ensures no details about the concrete or abstract semantics is leaked into the generic interpreter [13]. This design principle prohibits an operation `isZero` as in DAI. Instead, Sturdy provides a operation `ifZero(v: V, ifTrue: => R, ifFalse: => R): R`, where `ifTrue` and `ifFalse` are continuations. If both continuations must be run, Sturdy joins their results before moving on with the rest of the analysis. Sturdy uses a similar design for all operations that introduce uncertainty. For example, reading from a store is done by operation `read(a: Addr, ifFound: V => R, ifNotFound: => R): R`. We found the use of continuations in Sturdy excessive, making it harder to write and maintain the generic interpreter for Wasm. But can this be avoided?

In our framework, we have retained Sturdy's design principles to permit modular reasoning about components. While our framework does not attempt to support formal proofs, modular reasoning reemerges in the form of modular soundness propositions that can be used during testing. However, we significantly reduce the amount of continuations needed by encapsulating uncertain results in dedicated auxiliary data types: `JOption` and `JEither`. These data types provide standard operations such as `getOrElse`, `map`, and `flatMap`. For the concrete semantics, these data types behave identical to the standard `Option` and `Either` types, but their abstract semantics can encode uncertainty such as `LeftOrRight(l, r)`. Besides reducing the number of continuations needed, these types significantly improve the readability of component interfaces. For example, reading from a store has the simple signature `read(a: Addr): JOption[MayJoin, V]`.

[1] Link to source code repository omitted due to double-blind reviewing.

```
// DAI: k-CFA analysis of PCF, 6 components
ReaderT (FailT (StateT (NondetT (CacheT (FinMap0 Power0) ID))))

// Sturdy: k-CFA analysis of PCF, 8 components
ValueT (ErrorT (EnvT (FixT (ComponentT (StackT (CacheT (CallSiteT (->))))))))

// Sturdy: constant propagation of Wasm, 15 components
ValueT (JumpTypesT (OperandStackT (ExceptT (StaticGlobalStateT
  (MemoryT (SerializeT (TableT (FrameT (LogErrorT
    (FixT (ComponentT (StackT (CacheT (ControlFlowT (->)))))))))))))))
```

Figure 4 Deep transformer stacks as required by DAI and Sturdy impair the performance of the analyzers.

Eliminating the monadic transformer stack. Both DAI and Sturdy encode the generic interpreter in monadic style: The side effects triggered by the analyzed program are threaded through the monadic computation. And both frameworks use transformers to decompose effect handling into components. For example, in Figure 4 we show the transformer stacks used by DAI and Sturdy for a k-CFA analysis of PCF, as well as the transformer stack for our prototypical constant propagation analysis of Wasm implemented in Sturdy. This shows how the transformer stack grows considerably when analyzing complex languages.

Large transformer stacks are problematic because they impair the performance of the interpreter. Every monadic operation in the interpreter must traverse the entire transformer stack, slowing down interpretation considerably. Keidel et al. [12] measured this effect and showed that an interpreter on a transformer stack was 7756x slower than the same computation after exhaustive inlining of the entire stack. Thus, they argued that inlining allows us to enjoy modularity without regrets. While we concur in principle, this approach does not scale to complex languages unfortunately. For transformers stacks like the one for Wasm shown in Figure 4, the compiler exceeded 16 GB of memory while inlining and ultimately failed to compile the program. Since a 7756x slower analysis is not feasible, we must find an alternative design to support modularly defined components.

In our framework, we follow an object-oriented design in representing independent components. Rather than stacking all components and threading their effect through the computation, we let each component manage and manipulate its own internal state. As usual in OO, the internal state is encapsulated in the component and hidden behind a public interface. For example, setting a global variable `globals.set(x, stack.popOrFail())` changes the internal state of `stack` and `globals`, which is observable through operations of the public interface, such as `globals.get`. Since components are not stacked, invoking a component's operation is a simple method call that does not involve any other components.

Only when joining effectful computations, all effect components must participate, each taking care of their own internal state. The generic interpreter defines an effect stack that determines the order in which effects are joined. For Wasm, we use the following effect stack:

```
val effectStack = EffectStack(List(stack, memory, globals, funTable, callFrame, except,
      failure))
```

Each abstract semantics of an effect component must implement `joinComputations(f)(g)`, which executes `f` and `g` on the current internal state and merges the two resulting states. We apply a common strategy to implement these joins:

1. Take a snapshot of the internal state.
2. Execute `f`, store the resulting state.

3. Restore the snapshot state.
4. Execute g, store the resulting state.
5. Join the two states in an effect-dependent manner.

Consider the following example program:

```
// locals before: 0 := 0; 1 := 10
(if (then (i32.const 25) (local.set 0)) (else (local.get 0) (local.set 1))
// locals after: 0 := (25 ⊔ 0); 1 := (10 ⊔ 0)
```

The then branch produces a call frame that still maps `0 := 25` and `1 := 10` unchanged. The else branch must operate on a copy of the original call frame and produce `0 := 0` unchanged and `1 := 0`, ignoring the manipulations done in the then branch. Finally, we join the resulting call frames, obtaining the result shown above. In the next section, we show that analyses defined in our framework scale to real-world programs.

7 Evaluation

Section 5 has already demonstrated how our approach enables the modular construction of Wasm analyses. In this section, we present empirical results that attest (i) the concrete interpreter is correct, (ii) the static analyses are sound with respect to the concrete interpreter, and (iii) the type, constant, and taint analyses yield relevant results.

Correctness of concrete interpreter. Establishing the correctness of the concrete interpreter is important, because the concrete interpreter provides ground truth for reasoning about the soundness of our analyses. Thus, any soundness result we may provide is only meaningful as long as the concrete interpreter itself is a true implementation of the Wasm specification. In particular, our analyses and the concrete interpreter share the generic interpreter, which must be correct. In fact, if there was a bug in the generic interpreter, this bug would *not* trigger a soundness violation, because the concrete interpreter would exhibit the same incorrect behavior. Therefore, establishing the correctness of the concrete interpreter is paramount.

To this end, we ran our concrete Wasm interpreter against the official test suite from the Wasm specification.[2] The test suite consists of 16481 assertions, testing the correct behaviour of the Wasm interpreter. This testing revealed several bugs in our implementation, all of which we fixed. For example, we found indexing errors in the linear memory and several subtle bugs concerning floating-point operations. Our concrete Wasm interpreter now passes the complete test suite.

Soundness of static analyses. Only sound analyses can be used to inform program optimizations without jeopardizing the program's semantics. Since we want to conduct performance optimizations and reduce the size of Wasm binaries, we must ensure our analyses are sound. To this end, we tested soundness of our analyses against the concrete interpreter. Our platform allows us to implement soundness propositions for each value and effect component modularly. Value components implement an abstraction function that lifts the canonical concrete value representation into the abstract domain, using a partial order on the abstract domain to determine sound approximation. Effect components implement a soundness proposition that relates the internal state of the canonical effect implementation to their own internal state. That is, we not only check the final value computed by an analysis,

[2] `https://github.com/WebAssembly/spec/`

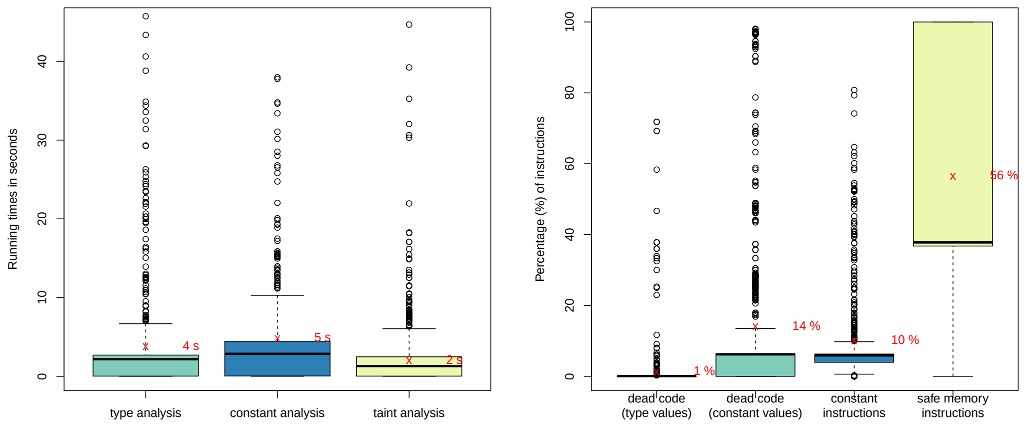


Figure 5 Running times in seconds (left) and analysis results in % of instructions (right) for analyzing each of the 1458 WasmBench binaries. We show the average times and results in red.

but also the final state of the linear memory and other effect components. An analysis then simply composes the soundness propositions of its components.

We tested the soundness of our analyses against the concrete interpreter on the test suite from the Wasm specification. Specifically, we ran the analyses and the concrete interpreter simultaneously and tested analysis soundness after every single assertion. This uncovered several bugs. For example, we initially defined integer division `Top / Top = Top`, which neglects division-by-zero errors and should yield `Top` $\sqcup$ `fail(...)` instead. We were able to fix all soundness bugs, so that we are confident the abstract interpreters are sound with respect to the concrete interpreter.

Large-scale evaluation. To assess the usefulness and performance of our analyses, we applied them to the programs collected by others in the WasmBench benchmark suite. WasmBench [10] contains 8461 unique Wasm binaries collected from various sources, including github, NPM, and by crawling websites. Out of these, we had to ignore 7003 binaries that failed to validate, 6354 of which due to unresolvable imports of modules not collected by the benchmark suite. Since WasmBench collects individual binaries rather than applications, we have no principled means of finding the right module. Another 607 modules were rejected due to invalid memory page size information. For each binary of the remaining 1458 binaries, we run our analyses using the most general client described in Section 5.4, so that the analysis results soundly approximate any potential usage of the module.

We measured the running times after a warm-up phase. We cancelled analysis runs after 60 seconds, which yielded between 196 and 200 timeouts per analysis. This timeout was chosen for pragmatic reasons: To limit the overall time required to run the experiment, which finishes in a little over 7 hours. Figure 5 shows the running times of the remaining analysis runs, representing 87% of the binaries. On average, the type analysis finishes in 4s, the constant analysis in 5s, and the taint analysis in 2s. The taint analysis is faster because it does not construct a call graph. We note that 81% of all type and constant analysis runs finish in 10s or less (including those runs that timed out), as do 85% of all taint analysis runs.

Figure 5 shows the percentage of instructions our type-based dead code, constant-based dead code, constant propagation analysis, and taint analysis identified. We count an instruction as dead if it is unreachable or, in case of blocks and loops, if they are never targeted by a jump. Such dead instructions can be safely eliminated from a Wasm binary. This reduces the binary size and saves bandwidth if the binary is sent over the network. Unsurprisingly, our baseline type analysis cannot find much dead code. However, even a simple constant propagation analysis can already reduce binaries by 14% on average. Note that the dead code this analysis identified was missed by other compilers, as many of the binaries stem from deployed packages and websites. The constant analysis also identifies 10% of instructions as computing constant results. This excludes instructions like `i32.const` of course. Constant instructions can be replaced by such `const` instructions. Due to our modular architecture, analysis developers can focus on improving one aspect of the analysis at a time to increase the optimization potential further.

Finally, the goal of the taint analysis is to track the data flow of tainted values and detect if tainted values can reach critical program points. Our taint analysis defines user input and results of calling host functions as tainted and detects potential security risks if tainted values are used as memory addresses. Protecting the memory is important because many compilation schemes targeting Wasm use the memory to embed critical infrastructure of the source language's runtime system [19]. For example, some runtime systems manage their own call stack in the memory, which thus is not protected from the user. If we can show that the user cannot access or and manipulate the memory shape, this means that the runtime system cannot be tampered with this way. Consequently, we consider a memory access to be safe if the analysis can guarantee that a tainted (user-influenced) value cannot be used as an address. On average, our analysis finds 56% of all memory accesses to be safe. Out of the 1458 Wasm binaries, our analysis shows 28% to be completely safe, meaning they only contain safe memory accesses. This analysis is fairly simple still and, for example, does not support any sanitization of tainted values, which should further improve the analysis results.

Comparison with the industry standard. While we compare to related work in the subsequent section, we thought it important to validate our approach empirically in comparison to the industry standard. The de-facto industry standard for Wasm code optimization is Binaryen[3], a C++ library that provides its own Wasm IR and implements about 100 optimization passes in its `wasm-opt` tool. This includes whole-program constant propagation and dead code optimizations, although the details and limits of the underlying analyses are not clearly documented. This begs the question: Can our approach compete with Binaryen, an industry standard for Wasm optimization developed by more than 140 contributors.

We answer this question quantitatively by running the optimizer of Binaryen on all WasmBench binaries that we successfully optimized. Binaryen transforms the Wasm code into its own IR, optimizes that IR, and translates it back into the Wasm binary format. We configured Binaryen using the `-Oz` flag, which aggressively optimizes for code size. We compute the number of *eliminated instructions* by loading the original and the optimized module and subtracting their instruction counts. We then compare this number to our constant analysis, where each dead or constant instruction counts toward the *eliminated instructions*. Figure 6 shows the results of our experiment.

Our experiment clearly shows that our approach outperforms Binaryen in terms of precision, eliminating twice as many instructions on average. While further investigation is necessary to understand where exactly our approach wins compared to Binaryen, note that we have built a generic framework for Wasm analyses. In particular, constant propagation

[3] `https://github.com/WebAssembly/binaryen`

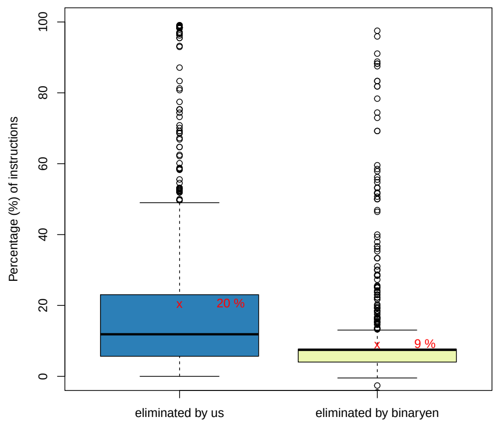


Figure 6 Comparing our approach to Binaryen, the industry standard for Wasm optimizations.

is a simple abstract domain and we may expect far better precision by using intervals or even relational abstract domains. Our framework is designed to accomodate those future improvements. In terms of performance, Binaryen only takes 0.1s on average, where our callsite-sensitive constant propagation analysis takes 4.8s on average. This is to be expected, given that our analysis lies in a different complexity class.

One important threat to validity of this experiment is that our analyses do not actually rewrite Wasm binaries. Instead, we count the number of instructions that were detected as dead or constant. We believe this is fair, since dead instructions can be dropped for sure and the constant instructions can be removed by propagating the constant value. Actually, we penalizes our own approach because in `const 1; const 2; add`, we only count the last instruction as eliminable, while Binaryen removes all three of them. We hope to integrate our analysis into a framework like Binaryen in future work to realize optimizations based on our analysis results.

8 Related Work

Our work investigates how to develop modular static analyses for Wasm using abstract definitional interpreters. We have already compared to prior approaches of abstract definitional interpreters in Section 6 in detail. In this section, we discuss how our work relates to prior work on Wasm, X86 assembly, and JVM bytecode.

Stiévenart and Roover [27] designed the first static taint analysis *wassail* for Wasm using a compositional approach. In particular, they analyze each function in isolation and compute a summary of the taint information of the following form:

```
function 8: stack: [l0,l1], globals: [g0;l1], mem: g7
```

This example summary means that the Wasm function with id 8 may store the variables `l0`, `l1` on stack, may store the variables `g0`, `l1` as globals, and variable `g7` in the linear memory. In a second step, they combine the summaries of multiple functions in bottom-up order of the call graph to compute the complete analysis result. While compositional analyses are known to scale better, they are also less precise than whole-program analyses. There are two places where our whole-program taint analysis is more precise than wassail's compositional taint analysis. First, wassail does not resolve indirect calls precisely. In particular, an indirect call reads the function index from the stack, which is not approximated by wassail. Instead, wassail resolves an indirect call to all functions which have a matching type. This may be especially imprecise for common function signatures such as `F64 -> F64`. In contrast, our constant taint analysis approximates the stack and is able to resolve indirect calls precisely in case the function index is a constant. Second, wassail does not approximate the layout of wasm's linear memory precisely. In particular, wassail returns all taint variables stored in memory on every load instruction. In contrast, our constant taint analysis approximates the layout of wasm's linear memory more precisely. Specifically, our memory abstraction separates taint values stored at constant addresses from taint values stored at non-constant addresses. This increases the precision of load instruction with a constant address.

Wasp[4] is a C++ library for performing simple static analyses on Wasm code. It offers methods to dump specific parts of a module (e.g., all functions) and to compute a function's call graph, control-flow graph, and data-flow graph. In contrast to our work, Wasp is not designed to implement more sophisticated analyses for Wasm but rather as a tool making it easy to work with Wasm modules. In particular, Wasp does not consider abstract domains to approximate values and thus, by and large, yields results equivalent to our type analysis. But, as our evaluation showed, even simple value domains such as constant propagation improve the precision of analyses significantly: The type analysis only found 1% of dead instructions on average, whereas we were able to prove 14% of instructions are dead using an abstract domain for constant propagation. This is out of reach for Wasp.

Wasabi [20] is a general purpose framework for implementing *dynamic* analyses for Wasm, which can be implemented using a high-level JavaScript API. The framework then instruments the Wasm binary to call these JavaScript analysis functions. Dynamic analyses are used in different contexts than static analyses. While analyses for security (e.g., a taint analyses) may be performed both statically and dynamically, compiler optimizations entail the use of a static analysis. Hence, the focus of their work is orthogonal to ours and explores a different part of the design space.

Watt et al. [33] developed two formal semantics for Wasm in the Isabelle and Coq proof assistants. These formal semantics can be used to prove properties about Wasm programs. However, these proofs require a high amount of manual effort and expertise in contrast to static analyses, which are automized.

Static analysis of x86 assembly code [3, 6, 15] faces several challenges summarized in the PhD thesis of Kinder [14]. For example, unstructured control-flow with `goto`'s and long jumps with dynamic jump target complicate the construction of a control-flow graph [16, 23]. Furthermore, x86 programs store their code alongside the data during the execution, which makes it harder for static analyses to differentiate between them [32]. This also allows x86 programs to modify their own code during execution, which poses a severe challenge for static analyses [29]. In contrast, Wasm prevents these problems with a stricter language design. In particular, Wasm is statically-typed, features only structured control-flow and clearly separates between code and data, which makes it impossible for Wasm programs to modify their own code [9]. The stricter language design of Wasm lowers the bar for implementing static analyses and improves their precision compared to x86 analyses.

Many static analysis frameworks for Java target JVM bytecode [1, 4, 26], the assembly code that underlies the Java Virtual Machine [21]. However, JVM bytecode poses a challenge to static analyses, because of its implicit dataflow and due to the use of a stack. Vallee-rai and Hendren [31] solved this problem by compiling JVM byte code to *Jimple*, a simpler

[4] `https://github.com/WebAssembly/wasp`

three-address code. Jimple is easier to analyze than JVM bytecode, because the addresses relieve from having to extract dataflow information from the stack. Since its inception, Jimple has become the defacto standard for analyzing JVM bytecode and is used by popular Java analysis frameworks such as Doop [24, 8] and Soot [30, 5, 2, 25]. In contrast, we show that abstract definitional interpretation can be used to analyze Wasm code directly, without requiring another intermediate representation, such as Jimple. This is a key advantage of abstract definitional interpretation.

Koren [17] presented an integrated development environment for Wasm that can be used to develop high-performance and latency-sensitive Wasm applications for the internet of things. Such an IDE would benefit from static analyses built with our modular platform, as static analyses can provide valuable feedback to the developer about low-level and hard to understand Wasm programs.

Lehmann et al. [19] and Stiévenart et al. [28] investigated the security risk of compiled Wasm programs. In particular, C applications compiled to Wasm reexperience security problems that are well known and fixed in the native C compiler. More specifically, the compiled C programs are vulnerable to stack and heap-based buffer overflow attacks. These vulnerabilities can be detected by static analyses for Wasm code.

9 Conclusion

In this work, we developed the first whole-program control and data-flow analyses for Wasm. It is important that we understand how to analyze Wasm programs for enabling optimizations and to find bugs and vulnerabilities. Our analyses lay the foundation for that as they scale to real-world programs, where we find 14% of all Wasm instructions are dead code, 10% of all instructions can be replaced by constants, and 56% of all memory accesses are safe against tampering.

Our analyzers are based on two fundamental contributions this paper makes. First, we present a decomposition of the Wasm semantics into 19 language-independent components that abstract different aspects of Wasm. This decomposition allowed us to develop static analyses modularly, which was essential for limiting the complexity of the implementation and the development effort. Second, we show how abstract definitional interpretation can be used to implement modularly defined static analyses for complex languages at scale. We explained how our new framework for abstract definitional interpretation eliminates the inefficiencies of prior frameworks, and why that was crucial for scaling to complex languages and real-world programs. The lessons learned for building abstract definitional Wasm interpreters can certainly be transferred.

References

1 Watson libraries for analysis (wala). URL: `http://wala.sf.net/`.

2 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. ACM, 2014. `doi:10.1145/2594291.2594299`.

3 Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*

2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings, volume 2985 of *Lecture Notes in Computer Science*, pages 5–23. Springer, 2004. doi:10.1007/978-3-540-24723-4_2.

4 Roberto Barbuti, Nicoletta De Francesco, and Luca Tesei. An abstract interpretation approach for enhancing the java bytecode verifier. *Comput. J.*, 53(6):679–700, 2010. doi:10.1093/comjnl/bxp031.

5 Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and soot. In Eric Bodden, Laurie J. Hendren, Patrick Lam, and Elena Sherman, editors, *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012*, pages 3–8. ACM, 2012. doi:10.1145/2259051.2259052.

6 Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. Static detection of vulnerabilities in x86 executables. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 269–278. IEEE, 2006.

7 David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. Abstracting definitional interpreters (functional pearl). *PACMPL*, 1(ICFP):12:1–12:25, 2017.

8 Neville Grech and Yannis Smaragdakis. P/taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA):102:1–102:28, 2017. doi:10.1145/3133926.

9 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with webassembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 185–200. ACM, 2017. doi:10.1145/3062341.3062363.

10 Aaron Hilbig, Daniel Lehmann, and Michael Pradel. An empirical study of real-world webassembly binaries: Security, languages, use cases. In *WWW: The Web Conference*, pages 2696–2708. ACM / IW3C2, 2021.

11 David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 51–62. ACM, 2010.

12 Sven Keidel and Sebastian Erdweg. Sound and reusable components for abstract interpretation. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360602.

13 Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. Compositional soundness proofs of abstract interpreters. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–26, 2018.

14 Johannes Kinder. *Static analysis of x86 executables (Statische Analyse von Programmen in x86-Maschinensprache)*. PhD thesis, Darmstadt University of Technology, 2010. URL: http://tuprints.ulb.tu-darmstadt.de/2338/.

15 Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 423–427. Springer, 2008. doi:10.1007/978-3-540-70545-1_40.

16 Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In Neil D. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2009. doi:10.1007/978-3-540-93900-9_19.

17 István Koren. A standalone webassembly development environment for the internet of things. In Marco Brambilla, Richard Chbeir, Flavius Frasincar, and Ioana Manolescu, editors, *Web Engineering*, pages 353–360, Cham, 2021. Springer International Publishing.

18 Erik Krogh Kristensen and Anders Møller. Reasonably-most-general clients for javascript library analysis. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 83–93. IEEE / ACM, 2019.

19 Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann`.

20 Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing webassembly. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 1045–1058. ACM, 2019. `doi:10.1145/3297858.3304068`.

21 Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

22 Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. New kid on the web: A study on the prevalence of webassembly in the wild. In Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*, volume 11543 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2019. URL: `https://doi.org/10.1007/978-3-030-22038-9_2`, `doi:10.1007/978-3-030-22038-9\_2`.

23 Minh Hai Nguyen, Thien Binh Nguyen, Thanh Tho Quan, and Mizuhito Ogawa. A hybrid approach for control flow graph construction from binary code. In Pornsiri Muenchaisri and Gregg Rothermel, editors, *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 2*, pages 159–164. IEEE Computer Society, 2013. `doi:10.1109/APSEC.2013.132`.

24 Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, 2015. `doi:10.1561/2500000014`.

25 Johannes Späth, Karim Ali, and Eric Bodden. Ide^{al}: efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA):99:1–99:27, 2017. `doi:10.1145/3133923`.

26 Fausto Spoto. The julia static analyzer for java. In Xavier Rival, editor, *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2016. `doi:10.1007/978-3-662-53413-7\_3`.

27 Quentin Stiévenart and Coen De Roover. Compositional information flow analysis for webassembly programs. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 28 - October 2, 2020*, pages 13–24. IEEE, 2020. `doi:10.1109/SCAM51674.2020.00007`.

28 Quentin Stiévenart, Coen De Roover, and Mohammad Ghafari. The security risk of lacking compiler protection in webassembly, 2021. `arXiv:2111.01421`.

29 Tayssir Touili and Xin Ye. Reachability analysis of self modifying code. In *22nd International Conference on Engineering of Complex Computer Systems, ICECCS 2017, Fukuoka, Japan, November 5-8, 2017*, pages 120–127. IEEE Computer Society, 2017. `doi:10.1109/ICECCS.2017.19`.

30 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In Stephen A. MacKay and J. Howard Johnson, editors, *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13. IBM, 1999. URL: `https://dl.acm.org/citation.cfm?id=782008`.

31 Raja Vallee-rai and Laurie Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.

32 Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani M. Thuraisingham. Differentiating code from data in x86 binaries. In Dimitrios Gunopulos, Thomas Hofmann, Donato Malerba, and Michalis Vazirgiannis, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2011, Athens,* *Greece, September 5-9, 2011, Proceedings, Part III*, volume 6913 of *Lecture Notes in Computer Science*, pages 522–536. Springer, 2011. `doi:10.1007/978-3-642-23808-6\_34`.

33 Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. Two Mechanisations of WebAssembly 1.0. In *FM 2021 - Formal Methods*, pages 1–19, Beijing, China, November 2021. URL: `https://hal.archives-ouvertes.fr/hal-03353748`.