Internship Report
From Jasmin to Wasm

Hugo Barreiro M2 MPRI
Supervised by Benjamin Grégoire and Manuel Serrano
Centre Inria d'Université Côte d'Azur, SPLiTS team

March 2026 — July 2026

General context

Jasmin is a dedicated environment for high-performance trustworthy cryptography. Its implementations are designed to be efficient, safe, correct, and secure. The Jasmin programming language combines high- and low-level constructs, allowing the developer to anticipate the generated assembly code. In order to reason rigorously about program behavior, its semantics is formally defined, enabling proofs of properties about these programs.

WebAssembly is a recent virtual machine whose objective is to provide a universal execution platform. Despite its name, it was designed as a versatile assembly language, much like the JVM, rather than specifically for the Web. One of Wasm's main strengths is its safe execution model, making it a promising target for all critical applications, and in particular cryptographic computations.

Research problem

The objective of this internship is to implement a new WebAssembly code generator in the Jasmin environment. Although other languages already offer this compilation target, integrating it into Jasmin allows taking advantage of its strengths, such as access to formal program verification, the efficiency of generated code, and high-level abstractions provided within a low-level language.

Using a WebAssembly backend enables implementing safe and efficient programs for the Web, particularly cryptographic algorithms. Thus, it becomes possible to leverage Jasmin to verify all necessary properties thanks to extraction towards EasyCrypt, a proof assistant specialized for cryptographic proofs.

Since Jasmin is a general-purpose language and Wasm is a universal platform, nothing prevents using them in other contexts. For instance, it is conceivable to write image processing algorithms in Jasmin in a more pleasant, or even more efficient, way than in JavaScript for web applications. Furthermore, the resulting WebAssembly cryptographic programs could interface with other languages, for example OCaml via Wasm-of-OCaml. This approach allows offloading critical sections to Jasmin and integrating them into other ecosystems.

Your contribution

We have developed, in OCaml, a complete backend to generate Wasm code. We have also integrated specific primitives into it, notably vectorized instructions required to adapt Jasmin programs orig-

inally designed for the x86-64 architecture. To achieve this, we had to formally define, in Rocq, a minimal version of the WebAssembly architecture in order to connect the compiler's frontend to the new backend. Finally, we extended the Jasmin language by introducing the reference type to enable using the WasmGC extension with the generated programs.

The generated code turns out to be particularly efficient: the various benchmarks, carried out on practical cryptographic programs, show performance comparable to that of the x86-64 backend. For identical source codes, both targets perform similarly. Programs originally optimized for x86-64 suffer only an overhead of around 14% to 24% after minimal compatibility modifications. The gap drops to between approximately 6% and 15% only after naive specialization for WebAssembly.

Additionally, we wrote Jasmin programs specialized for the Wasm backend to perform image processing. To achieve maximum performance for these programs, we needed to add other, less common, WebAssembly-specific instructions to Jasmin. These measurements highlight a significant performance gain compared to JavaScript when intensive computations are delegated to the Wasm code produced by Jasmin.

Arguments supporting its validity

Since this backend is written in OCaml and not in Rocq, its functional correctness cannot yet be formally proven. Nevertheless, we set up a test suite, notably including unit tests for instructions specific to the WebAssembly architecture. For Jasmin instructions, their translation to Wasm is validated by differential testing against the x86-64 architecture, whose functional correctness is formally proven.

Summary and future work

We implemented an OCaml backend to generate Wasm code. It supports the full Jasmin instruction set, enriched with several primitives specific to WebAssembly. The benchmarks demonstrate satisfactory performance, notably only about 10% to 15% overhead compared to x86-64 code. Further measurements indicate that Jasmin code specialized for Wasm constitutes a promising alternative to replace intensive computations written in JavaScript.

Several future directions can still be pursued. In the short term, reimplementing the backend in Rocq would allow formally certifying its correctness. Another direction consists in exploiting the constant-time extension of WebAssembly to preserve constant-time on the generated code, before studying the impact of JIT compilation on this property.

Contents

1 Introduction

The main objective of this internship is to add a new backend to the Jasmin compiler that generates WebAssembly code.

1.1 Motivation

Implementing cryptographic algorithms presents major challenges. In addition to being particularly efficient so as not to slow down any critical application, implementations must also be correct in order to avoid compromising sensitive data. Ensuring both aspects is difficult because optimizing a program often complicates its analysis. For example, web applications require high efficiency, but their implementation is hard to prove, especially when written in languages like JavaScript.

This is where the idea of combining Jasmin with WebAssembly comes in. This combination brings the performance of Wasm code together with the assurances provided by Jasmin, in particular the ability to prove the correct behavior of the resulting WebAssembly program. Thus, this code could be used on the Web to reconcile efficiency and safety.

This new backend also facilitates interfacing the code with high-level languages such as OCaml. Indeed, Jasmin offers a more general-purpose framework beyond cryptography, with finer control over the final assembly code. This provides an opportunity to bring additional support to languages where low-level concerns are usually abstracted away.

In this report, we present how to implement a new backend in Jasmin to generate efficient Wasm code that can be used in practice. We will take advantage of the structures already established by this platform.

1.2 Intuition

The goal is to rely on the existing structures of the Jasmin platform rather than reimplementing them entirely. In particular, most of the compilation passes present in the frontend can be reused. However, this is not really possible for the backend. Indeed, a WebAssembly program has a structure that differs significantly from the other assembly languages supported so far by Jasmin, such as x86-64, RISC-V, or Arm M4. Wasm is a stack machine with linear memory whose code, based on structured control flow operators, is dynamically compiled. In contrast, the other architectures are register machines with physical memory whose code, based on labels and `goto` statements, is statically compiled.

First, an architecture for WebAssembly must be defined in Rocq, which initially can be more or less minimal. This consists in declaring essential information about Wasm, such as the data pointer size, register types, flag types, calling convention, certain instructions, etc. From then on, it becomes possible to reuse the existing frontend to use the transformed Jasmin program as input for the future backend.

Within the backend itself, compilation passes that are more specific to the WebAssembly architecture can be performed to further transform the Jasmin program. Once sufficiently prepared, the Jasmin program can be translated into Wasm. This task consists in identifying instructions that can be translated in a sufficiently predictable manner to preserve Jasmin's philosophy, which guarantees a nearly direct correspondence between the source program instructions and the generated assembly code.

To ensure the efficiency of the resulting WebAssembly programs, their performance is compared

against that of other backends for the same realistic cryptographic programs written in Jasmin. A differential test suite is also set up with another backend whose correctness is formally proven, to strengthen confidence in the correctness of the Wasm backend.

2 Context

This section presents the challenges associated with implementing cryptographic algorithms, and then introduces Jasmin and WebAssembly by detailing their specificities, definitions, and potential applications.

2.1 Implementation of Cryptographic Algorithms

Cryptography is a critical domain for several reasons. First of all, implementations of these algorithms are massively used, both locally and online. Their efficiency is therefore critical so as not to slow down applications. Second, they form the foundation of security protocols; it is thus crucial to guarantee their correctness, since a single bug would compromise the entire system. Finally, although algorithms may be cryptographically secure, their implementations are not necessarily so, as they are vulnerable to side-channel attacks. To resist such attacks, hardware constraints must be taken into account.

Consequently, the question of their practical implementation arises. Using structured languages is problematic because optimizations performed by compilers break resistance to side-channel attacks. Indeed, developers commonly use masking patterns to prevent secret data from influencing program branching or memory access: this is known as the constant-time property. Unfortunately, optimizing compilers manage to identify these patterns and replace them with standard structures, thereby destroying the protection provided. As a result, developers often end up implementing critical parts of algorithms directly in assembly to retain full control over the final code. However, writing directly in assembly presents major drawbacks: this tedious task complicates maintenance. Moreover, their inherent lack of structuring and abstractions makes reasoning about code particularly difficult. To address these challenges posed by cryptography, a new language was designed: Jasmin.

2.2 Jasmin

Jasmin [Almeida et al., 2019, Barthe et al., 2021, Almeida et al., 2024] is a dedicated environment for high-performance trustworthy cryptography. Its implementations must be efficient, safe, correct, and secure. Jasmin is a low-level language providing fine control over the generated assembly, allowing the generation of efficient code while preserving security properties. Furthermore, it offers high-level constructs such as variables, functions, arrays, and loops, ensuring good program structuring. Finally, its simple and clear formal semantics, combined with these abstractions, facilitates reasoning about programs, unlike traditional low-level languages. Thus, the Jasmin compiler produces predictable assembly code while guaranteeing that the use of high-level abstractions incurs no run-time cost.

Since Jasmin is developed and verified in Rocq, several key properties of the compiler are formally guaranteed. In particular, it is proven that the semantics of the source program is preserved by the generated code. Moreover, the functional correctness of every Jasmin program is provable thanks to an extraction mechanism to EasyCrypt, a proof assistant specialized for cryptographic proofs. Thus, any property established on the source code is intrinsically preserved during compilation. Finally, Jasmin's type system guarantees that if a program is well-typed, then it is constant-time, meaning it resists side-channel attacks.

Although well-suited for cryptographic implementations, Jasmin remains primarily a general-purpose language, which allows using it in other contexts, notably by interfacing it with other languages through its various backends, all while benefiting from its advantages, such as using precise assembly instructions within a structured language or the ease of proving properties on generated programs.

2.3 WebAssembly

WebAssembly [Haas et al., 2017, Youn et al., 2024], also known as Wasm, is a recent virtual machine whose objective is to provide a universal execution platform, currently offering compilation from C, OCaml, and Rust. Despite its name, WebAssembly is not limited to the Web: it is a virtual machine with a versatile assembly language, much like the JVM [Lindholm et al., 2021].

The provided language is strongly typed, memory-safe, executes in a sandboxed environment, and only supports structured control flow operators such as loops or functions. Wasm stands out from other virtual machines due to its source-language agnosticism and its proximity to hardware platforms, requiring only a minimal execution runtime, which makes it a portable virtual machine easily interfaced with web browsers in particular.

The security of its execution model is a major asset: strict typing rules out undefined behaviors, sandboxed isolation prevents illegitimate access to host memory (immunizing the system against buffer overflows), and any interaction with the operating system requires explicit permission. This makes it a promising target for all critical applications, and in particular cryptographic computations. In addition, the SIMD extension introduces 128-bit vectorized instructions, reinforcing its compatibility with high-performance cryptography where operation parallelism is essential.

WebAssembly features many other extensions, such as WasmGC, which enables efficient interfacing [Serrano and Findler, 2025] with high-level languages like Dart, Haskell, OCaml, Ruby, or Scheme through the introduction of a garbage collector. WasmGC was also recently added to version 3.0 of the WebAssembly specification. There is also CT-Wasm [Watt et al., 2019], which guarantees that WebAssembly programs are indeed constant-time.

3 The Jasmin Compiler

This section explains the objective of the Jasmin compiler and details its language as well as its architecture.

3.1 Objective

The primary objective of the Jasmin compiler is to offer a language that guarantees strict control over the generated assembly while providing high-level constructs that facilitate implementation and formal proof compared to different assembly languages. To achieve this, the language allows, among other things, choosing its instructions (Jasmin instruction or assembly instruction) and the type of memory used (register or stack).

To guarantee this predictability, the compiler refrains from performing any heavy code transformations. Thus, a Jasmin instruction should morally correspond to an assembly instruction. Emulating certain behaviors is not acceptable if it degrades program efficiency (in time or space) compared to using native instructions.

There is a fundamental difference between valid Jasmin programs (those that have semantics) and

Jasmin programs that can be compiled. Indeed, since the semantics is independent of target architectures, some architectures may lack instructions capable of natively expressing certain valid constructs. In such cases, the compiler prefers to fail. It is thus common for valid programs not to compile in order to respect predictability and strict control over the generated assembly. Jasmin's philosophy prioritizes sacrificing portability for efficiency to rival native code. Therefore, it is recommended to maintain, for each algorithm, a specialized version for each architecture.

3.2 Language

Jasmin has several integer types. On the one hand, the `int` type represents infinite-precision integers, whose value must be statically known. On the other hand, integer types of 8, 16, 32, 64, 128, and 256 bits accept arbitrary expressions. Depending on the target architecture, certain integer sizes may not be available due to platform limitations. Conversion operations between different integer types can be performed. To cast to a higher-precision integer, the conversion must be explicit in order to specify whether the user intends to perform a sign extension or a zero extension, two operations that can yield different results. Conversely, to cast to a lower-precision integer, the conversion is implicit since there is exactly one way to perform it. Integers support standard signed and unsigned arithmetic operations, as well as usual bitwise and logical operations.

The language also features arrays whose size must be known statically. Besides classic read and write operations, Jasmin allows explicitly choosing their memory placement (register or stack) using the *reg* and *stack* keywords. Two other types, *reg ptr* and *stack ptr*, morally act as pointers to arrays stored in registers or on the stack, respectively. Storing arrays on the stack can prove useful for relieving register pressure, thereby facilitating or enabling register allocation.

Jasmin programs are structured into non-recursive functions. The latter can be annotated with specific keywords. For instance, `inline` functions indicate that their body will be inlined at the call site. The `export` functions indicate that they can be called by external environments. Indeed, Jasmin is designed to be used by programs written in other languages, such as C, which can utilize the x86-64 code produced by Jasmin. Furthermore, to assist in structuring programs, the language offers `for` loops and `while` loops. The bounds of `for` loops must be statically known since they are fully unrolled. In contrast, `while` loops allow expressing loops of arbitrary length without unrolling.

In addition to standard instructions on integers and arrays, Jasmin allows direct use of assembly instructions specific to target architectures, referred to as *intrinsics*. Figure 1 illustrates the use of x86-64 *intrinsics* to perform an exact 128-bit addition efficiently. The *intrinsics* exploit addition with carry to use only two instructions, whereas without them, the user must perform a comparison to obtain the carry value, making the resulting code less efficient. While they enable expressing optimized computations very efficiently, they conversely make the program dependent on the target architecture, preventing compilation to other platforms. This feature reflects the core essence of Jasmin: extreme specialization at the expense of portability.

3.3 Architecture

The different compilation passes of Jasmin are illustrated in Figure 2. They are all formally verified, either by being directly written and proven in Rocq, or through a validator proven in Rocq that checks the accuracy of the pass result. The frontend encompasses passes ranging from the Jasmin language to the Stack intermediate language, while the backend collects target-architecture-dependent passes from Stack to ASM.

The frontend applies the necessary transformations to the typed Jasmin program to prepare for as-

```
export fn add128_adc(reg u64 a_low a_high b_low b_high)
-> reg u64, reg u64 {
  reg bool of cf sf pf zf;
  reg u64 r_low r_high;
  of, cf, sf, pf, zf, r_low = #ADD(a_low, b_low);
  of, cf, sf, pf, zf, r_high = #ADC(a_high, b_high, cf);
  return r_low, r_high;
}
```

```
export fn add128_no_adc(reg u64 a_low a_high b_low b_high)
-> reg u64, reg u64 {
  reg u64 r_low, r_high, carry;
  r_low = a_low + b_low;
  carry = 0;
  if (r_low < a_low) {
    carry = 1;
  }
  r_high = a_high + b_high + carry;
  return r_low, r_high;
}
```

Figure 1: Two Jasmin implementations of an exact 128-bit addition, one with x86-64 *intrinsics*, the other without

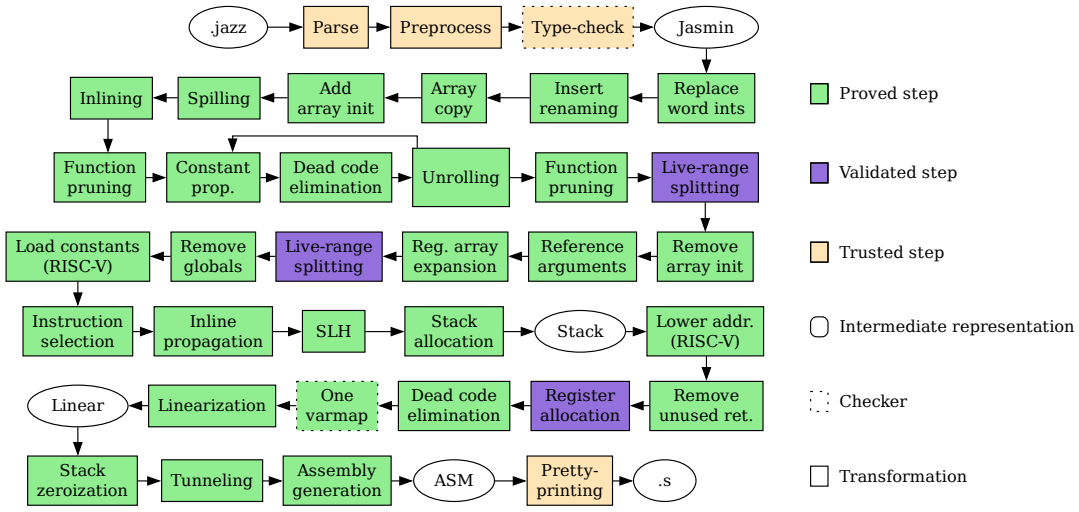

Figure 2: The compilation passes of Jasmin

sembly generation. It begins by replacing *word ints* (a machine integer type treated as `int` in the semantics) with machine integers, renaming certain function variables to comply with calling conventions, replacing pseudo-instructions `#copy`, `#spill`, and `#unspill` with their physical equivalents (real copies, spilling to stack or register), unrolling functions marked as *inline*, and deleting unused functions (such as *inline* functions that are not *export*). Next, a simplification phase repeatedly performs constant propagation, dead-code elimination, and `for` loop unrolling as much as possible. Functions that become unused are removed. The compiler continues with a form of SSA (Static Single Assignment) conversion to reduce variable lifetimes, insertion of *ptr* variables for function arguments and return values, transformation of each array cell into unique independent variables, followed by SSA conversion of the new variables. Jasmin instructions are then replaced by assembly instructions appropriate for the target architecture. Finally, variables marked as *inline* are propagated before proceeding to stack allocation. This crucial step consists in reserving stack areas to store global variables and local variables marked as *stack* (including arrays) for each function.

An analysis of this process highlights the effectiveness of constant propagation, which is maximized by the use of `int` variables, static `for` loops, and *inline* functions. These passes simplify or eliminate many intermediate instructions, thus optimizing the source program. Note that arrays are fully unrolled: each cell becomes an individual variable stored in registers or on the stack.

The backend finalizes compilation to assembly. It first removes unused return values from *export* functions and then proceeds to register allocation. During this critical step, all *reg* variables (includ-

ing those originating from unrolled arrays) are mapped to registers. This phase fails if the target architecture does not have enough registers to satisfy the program's constraints. Upon success, the compiler eliminates residual dead code, makes stack and argument management explicit, and translates structured control flow into labels and `goto` instructions. The latter are ultimately flattened prior to final assembly code generation.

Thus, even if a program is correctly typed, compilation may fail during instruction selection (if a primitive does not exist on the target architecture) or during register allocation (in case of register saturation).

4 Internship

This section presents the work carried out during the internship and the main results obtained. The code archive is available on GitHub.

4.1 The WebAssembly Backend

This section details the implementation of the Wasm backend. We first present the minimal definition of the WebAssembly architecture in Rocq, then the translation of Jasmin code to Wasm, implemented in OCaml.

4.1.1 Minimal Architecture Definition

The new WebAssembly backend is inserted just after stack allocation. The frontend passes benefit all architectures by optimizing the source code and managing arrays, in particular their allocation on the stack. Nevertheless, the passes performed by the other backends are of no interest in the case of Wasm. Unlike other architectures, WebAssembly is a stack machine relying on global or local variables and structured control flow. Register allocation and linearization are therefore unnecessary. We thus define a minimal architecture allowing the use of the frontend to insert the Wasm backend right after stack allocation.

In this architecture, there are no registers, only local or global variables. The memory is composed of one or more byte arrays. Function calls are performed with the `call` instruction. This makes it possible to specify that the data pointer is a 32-bit integer and that there are no registers, flags, or explicit calling conventions.

This step also involves declaring the WebAssembly instructions, i.e., their name, validity, type, semantics, and the nature of the arguments (arbitrary expressions or immediates). While these definitions are not strictly required for the OCaml backend, they are essential for its proof of correctness in Rocq and allow the exploitation of *intrinsics* in Jasmin programs.

The Jasmin language aims to be supported by all architectures, and it can prove restrictive for architectures with a particularly rich instruction set. Consequently, Wasm-specific instructions must absolutely be declared. For example, the most efficient implementations of cryptographic algorithms rely on vectorized instructions to perform parallel computations. Jasmin offers this kind of instructions, but some are not available. In the case of WebAssembly, it is therefore necessary to declare a few additional useful SIMD instructions, i.e., 128-bit vectorized instructions allowing parallel arithmetic computations that are impossible to express efficiently with the Jasmin language.

It becomes possible to exploit the compiler's frontend, which is therefore common to each architecture. Furthermore, certain frontend compilation passes may not be applied in the case of Wasm,

notably the one that introduces a renaming of certain variables to facilitate register allocation, which does not exist in WebAssembly.

4.1.2 Translation from Jasmin to Wasm

Before translating the program transformed by the frontend, a pass specific to the WebAssembly architecture must be added. The concept of implicit conversion is not a problem with other architectures since they also have it. However, this is not the case for Wasm, which is a strongly typed language, thus prohibiting, for instance, the addition of a 32-bit integer and a 64-bit integer. A prior cast of one of the operands is essential to obtain a valid operation, making it mandatory to make all conversions explicit.

To make these conversions explicit, the abstract syntax tree is traversed to retrieve the expected type for each argument of an operation and their current type, and then add a conversion if necessary. For example, if the addition expects two 32-bit integers and currently one of them is a 64-bit integer, then the conversion to a 32-bit integer must be added. Successive conversions are then flattened. For instance, a cast from 64 bits to 32 bits and then 16 bits becomes a direct cast from 64 bits to 16 bits.

Once this pass is complete, the transformed Jasmin program is then compiled to WebAssembly. The first step is to identify unsupported constructs. Because Jasmin must guarantee the predictability and strict control of the produced assembly code, the backend must fail if the user declares 8, 16, or 256-bit variables, types that do not exist in Wasm. Representing these variables with 32-bit integers would alter this predictability. Thus, all Jasmin instructions that do not have a sufficiently direct equivalent in WebAssembly cannot be compiled with this new backend. Furthermore, Wasm is not an isolated case, as similar configurations exist with the RISC-V and Arm M4 architectures.

Arithmetic shift operators illustrate another semantic divergence. In Jasmin, if the shift operand is too large, for example 33 for a 32-bit operator, then the result is equal to 0. In WebAssembly, this is not the case; the operand is evaluated modulo 32, so the result will be a shift of 1. Only shifts conforming to the Jasmin semantics can be compiled: the user must explicitly insert the modulo in the source code. This pattern will be recognized by the backend, which will compile it without calculating the modulo, since Wasm performs it implicitly. Figure 3 illustrates the compilation of these constructs.

```
export fn main(reg u32 x) -> reg u32 {
  reg u32 res;
  res = x << 1;
  res += x << 33;
  res += x << (x & 31);
  return res;
}
```

```
(module $shift
  (func $main (param $x.182 i32) (result i32)
    (local $res.194 i32) (local $res.195 i32) (local $res.196 i32)
    (local.set $res.194 (i32.shl (local.get $x.182) (i32.const 1)))
    (local.set $res.195 (i32.add (local.get $res.194) (i32.const 0)))
    (local.set $res.196 (i32.add (local.get $res.195)
      (i32.shl (local.get $x.182) (local.get $x.182))))
    (return (local.get $res.196))
  )
  (export "main" (func $main))
)
```

Figure 3: The compilation, from Jasmin to WebAssembly, of certain shift operations

It is also necessary to be able to recognize other patterns. Wasm only allows reading 8 or 16 bits at a certain memory address. Nevertheless, the result will be given by a 32 or 64-bit integer via a sign or zero extension. If the user, in their Jasmin program, decides to read 8 or 16 bits from memory and cast the result into a 32 or 64-bit integer, then we must be able to identify this situation in order to compile it to the corresponding WebAssembly instruction. Wasm also allows writing only 8 or 16 bits to a certain memory address from a 32 or 64-bit integer. A similar configuration arises when the

user decides to write only 8 or 16 bits to memory by casting their 32 or 64-bit integer. The backend must therefore recognize this precise pattern to compile it to the corresponding instruction. Figure 4 illustrates the compilation of the different cases mentioned.

```
export fn main(reg u32 in out) {
  reg u32 v1 v2;
  v1 = (32u)[:u8 in];
  v2 = (32s)[:u16 in + 8];
  [:u8 out] = (8u)v1;
  [:u16 out + 8] = (16s)v2;
}
```

```
(module $load
  (import "env" "memory" (memory $memory 1))
  (func $main (param $in.186 i32) (param $out.187 i32)
    (local $v1.199 i32) (local $v2.200 i32)
    (local.set $v1.199 (i32.load8_u (local.get $in.186)))
    (local.set $v2.200 (i32.load16_s (i32.add (local.get $in.186) (i32.const 8))))
    (i32.store8 (local.get $out.187) (local.get $v1.199))
    (i32.store16 (i32.add (local.get $out.187) (i32.const 8)) (local.get $v2.200))
  )
  (export "main" (func $main))
)
```

Figure 4: The compilation, from Jasmin to WebAssembly, of certain memory accesses

The *reg* variables, usually stored in registers, are placed in local variables in the generated code. In WebAssembly, it is not possible to manipulate the stack directly to store values there, so *stack* variables are stored in the Wasm linear memory: a space in which the stack is emulated as described in Figure 5. Because Jasmin functions are not recursive, the size of the activation records is known statically. Thus, along with global variables, they are stored in memory upon initialization of the WebAssembly module.

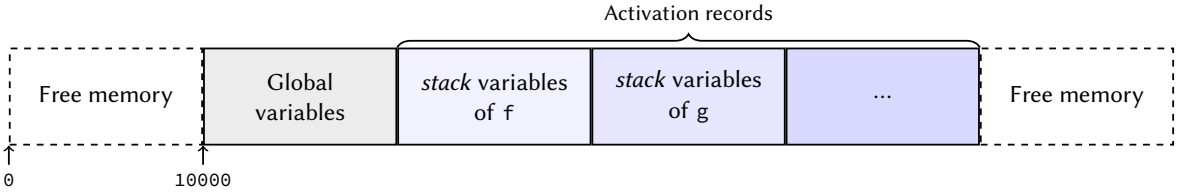

Figure 5: Organization of the WebAssembly memory for Jasmin programs

In other architectures, variables are stored on the stack. By design, when Jasmin is called by another program, this stack is separated from the external environment, allowing complete separation between the Jasmin source memory and the calling program's memory. However, this is not the case here with this memory management in Wasm. Indeed, the memory used by the source program can be modified by the caller. It is customary for the user to initialize or retrieve the module's memory to write information in order to communicate with the WebAssembly program.

A first complication is therefore found in the initialization of this memory, since the user has no way of knowing which parts of the memory will be used by the Wasm program and where they can use it safely. The second complication is that it makes the proof of correctness for the backend much more complex, since for other architectures the source memory and the external memory were necessarily separated. Fortunately, there is a solution, which has not yet been implemented. It would consist of marking memory accesses with a boolean indicating whether it is a source or external access. This would allow the introduction of two different and therefore separated memories in the generated WebAssembly program. Nevertheless, this solution requires updating all the compiler proofs, since they were designed for a world with a single memory from a semantic point of view, not two. This is the main obstacle that led us not to implement this solution during the internship.

4.1.3 The WasmGC Extension

Before the integration of WasmGC into the standard, one of the disadvantages of Wasm was that it did not natively have high-level data structures like arrays. Interfacing a WebAssembly program often required copying data into its linear memory, a potentially expensive operation. Cryptographic

applications thus suffered from two superfluous copies: the first when transferring bytes from the host language to the Wasm linear memory, and the second to retrieve the results. WasmGC eliminates these copies by introducing native structures and arrays in WebAssembly. The host language thus instantiates arrays that are directly usable by Wasm. The data is then copied and processed within these arrays, eliminating any redundant transfers. Regarding cryptographic algorithms, it becomes possible to store the bytes in a WebAssembly array as soon as the command line is read, avoiding any superfluous copying.

The integration of these features into Jasmin required the introduction of the reference type. Because references to Wasm arrays are semantically close to *reg ptr*, we exploit Jasmin's annotation mechanism to provide the compiler with more information about certain parts of the program. It is sufficient to annotate the *reg ptr* with a *ref* tag to guide the compiler during the various passes.

In practice, this is the case only during stack allocation. In Jasmin, an array is just a pointer on the stack, and accesses are memory reads or writes. In the case of references, we want to keep the array form since this is how they will be translated into WebAssembly. It is sufficient to add a boolean field to the internal data structure representing the *reg ptr*, indicating whether or not it is a reference. By initializing this field using the annotations attached to the various *reg ptr* in the program, during this compilation pass, the checks implemented on the *reg ptr* (for safety, for example) can be preserved, and then the stack transformation can be cancelled when it is actually a future Wasm reference.

To translate these arrays into WebAssembly, a new type must be declared for each type of array that exists and is compatible with Wasm, i.e., arrays of 8, 16, 32, 64, or 128-bit integers. Once this is done, it is natural to use WebAssembly read and write instructions to translate the reads and writes present in the source program. Like the 8 or 16-bit memory reads and writes, certain patterns introduced by references must be recognized. For example, reading from an 8 or 16-bit array returns a 32-bit integer in Wasm, so the result of the read must be cast to a 32-bit integer in the Jasmin source program in order to compile these reads to WebAssembly. This is also the case for writes to 8 or 16-bit arrays that expect a 32-bit integer; the 32-bit integer must be cast to an 8 or 16-bit integer in order to compile these writes to Wasm. One must not forget to export WebAssembly functions allowing the user to manipulate arrays from the external environment. For example, for each array type, we must provide a function to create an array, to read or write to it, and a final one to retrieve its size. Figure 6 illustrates the compilation of references for the mentioned patterns.

```
export fn main(
  #[ref] reg ptr u8[128] t1,
  #[ref] reg ptr u16[128] t2
  )
  ->
  reg ptr u8[128],
  reg ptr u16[128]
{
  reg u32 v1 v2;
  v1 = (32u)t1[0];
  v2 = (32s)t2[0];
  t1[0] = (8s)v2;
  t2[0] = (16u)v1;
  return t1, t2;
}
```

```
(module $ref
  (type $array_i8 (array (mut i8))) (type $array_i16 (array (mut i16)))
  (func $main (param $t1.186 (ref $array_i8)) (param $t2.187 (ref $array_i16))
              (result (ref $array_i8) (ref $array_i16))
    (local $v1.199 i32) (local $v2.200 i32)
    (local.set $v1.199 (array.get_u $array_i8 (local.get $t1.186) (i32.const 0)))
    (local.set $v2.200 (array.get_s $array_i16 (local.get $t2.187) (i32.const 0)))
    (array.set $array_i8 (local.get $t1.186) (i32.const 0) (local.get $v2.200))
    (array.set $array_i16 (local.get $t2.187) (i32.const 0) (local.get $v1.199))
    (return (local.get $t1.186) (local.get $t2.187))
  )
  (func $len_array_i8 (param $arr (ref $array_i8)) (result i32)
    (array.len (local.get $arr)))
  ;; Definition of $get_u_array_i8, $get_s_array_i8, $set_array_i8, $new_array_i8, ...
  (export "main" (func $main)) (export "len_array_i8" (func $len_array_i8))
  ;; Export of get_u_array_i8, get_s_array_i8, set_array_i8, new_array_i8, ...
)
```

Figure 6: The compilation, from Jasmin to WebAssembly, of certain read and write patterns in references

4.1.4 Testing Methods

Since the backend is not implemented in Rocq and therefore not yet proven at this stage, we decided to set up a test suite in order to have a little more certainty regarding the correctness of the backend. To do this, there are two main types of tests.

The first consists of unit tests for Wasm-specific instructions, so practically for SIMD instructions. These instructions cannot be compiled by other backends, so this is the only way to test them directly.

The second consists of differential tests with the x86-64 backend. The correctness of the latter is formally verified in Rocq. It is thus possible to compare, for arbitrary inputs, the results obtained by the two backends, knowing that the x86-64 backend necessarily returns the correct result. These differential tests follow two axes: on the one hand, Jasmin's `common` suite validates the generic instructions; on the other hand, the execution of real cryptographic algorithms allows for the indirect validation of SIMD instructions by replacing the x86-64 primitives with their WebAssembly equivalents.

4.2 Benchmark

This section presents the various benchmarks performed on the Wasm code generated by the new backend. We first compared its performance with that of the x86-64 backend, and then also tested the efficiency of interfacing Jasmin code compiled to Wasm with other languages.

4.2.1 Comparison with the x86-64 Backend

The backend provides us with a textual file in the `.wat` format. Once the WebAssembly code is obtained, it can be interpreted. However, for performance reasons, the usual approach is to execute it in an external environment. To do this, the textual `.wat` file must be compiled into a binary `.wasm` file. This processing can be entrusted to a tool like `wasm-as`, which will literally translate the instructions into binary, or `wasm-opt`, which has four optimization levels. With the latter, Wasm programs will be optimized before being translated into binary. The `.wasm` file is then connected to the external environment, the most famous being JavaScript. To integrate the WebAssembly program into JavaScript and execute it, one must rely on either Node.js (a JavaScript runtime environment) or directly on a browser. The Wasm code is no longer interpreted but is finally just-in-time compiled to x86-64 code.

The JavaScript code and, by extension, the WebAssembly code used, benefit from JIT (Just-In-Time) compilation to make the code highly efficient. A JIT will first compile the Wasm code to naive x86-64 code whose performance is very limited; nevertheless, at the same time, it launches a more complex analysis that will yield highly efficient x86-64 code. Once this analysis is finished, the naive code will be replaced by the efficient code. Furthermore, thanks to a monitoring system, the JIT analyzes the execution in real-time, allowing it to refine the optimized code obtained to make it even more efficient in the context in which it is used.

Node.js and Chromium-based browsers (Brave, DuckDuckGo, Google Chrome, Microsoft Edge, Opera, …) take advantage of the V8 engine, while Mozilla Firefox uses the SpiderMonkey engine. These are the two main JIT engines used in practice. During the benchmarks, we will try to determine which is the most efficient depending on the context.

In order to verify if the efficiency of the produced WebAssembly code rivals that of the x86-64 backend, we evaluated several realistic cryptographic algorithms written in Jasmin. Algorithms using

AVX instructions (SIMD in Wasm) are optimized and intended solely for x86-64, and therefore cannot be compiled directly to WebAssembly because they use x86-64 *intrinsics*. The others are reference versions, meaning they use very few x86-64 instructions (except Gimli, which uses none).

To successfully compile these algorithms to Wasm, we simply replaced the x86-64 instructions with Jasmin or WebAssembly instructions: these are the minimal versions. However, since these programs were initially intended for x86-64, some implementation choices are not at all optimal for Wasm. We notably find addresses stored on 64 bits (which will have to be truncated to 32 bits in Wasm), *stack* variables to make register allocation feasible even though there is an unlimited number of local variables in WebAssembly, or saving variables on the stack to facilitate register allocation even though there is no register allocation for Wasm. Thus, we rewrote another version incorporating these architectural optimizations to obtain better performance in WebAssembly: these are the optimized versions.

To measure the performance of the x86-64 and Wasm versions, we repeat the algorithms a sufficiently large number of times to obtain computation times between 5 and 10 seconds for each version. Additionally, we repeat the experiment 20 times to mitigate the variance between each measurement. We also fix the processor core to be used in order to stabilize the measurements as much as possible. Finally, before performing the actual measurements, we iterate 10% of the repetitions in the void to avoid the initial unstable times and allow the JIT to obtain the optimized version of the program.

Figures 7 and 8 present, respectively for the minimal and optimized versions, the results of the benchmark by comparing the ratio between the best execution time of the Wasm backend (among the different optimization levels and JIT engines) and the execution time of the x86-64 backend.

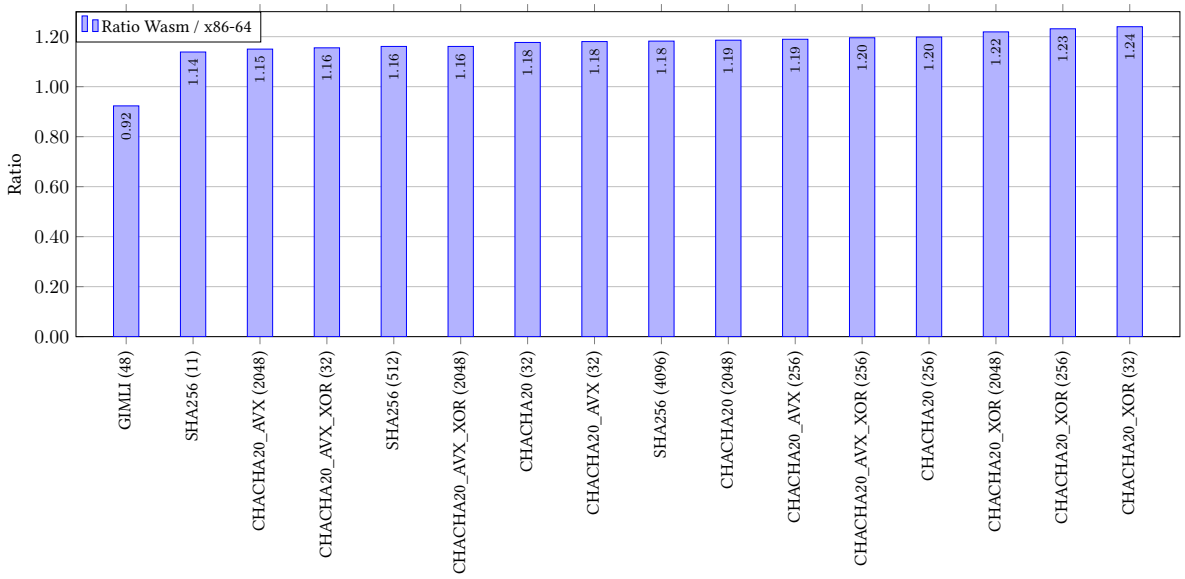

Figure 7: Ratio of execution time on the minimal versions between Wasm and x86-64

First, let us observe that the WebAssembly backend is slightly more efficient by about 8% for Gimli, which is the only program that is exactly identical for both backends. This efficiency can be explained by the lower number of optimizations received by x86-64. Indeed, the Wasm backend receives the Jasmin compiler's optimizations (like x86-64) but can also receive the optimizations from `wasm-opt` as well as those from the JIT. Thus, on the same program (at the Jasmin level), the code generated by the WebAssembly backend can be more efficient at execution. Next, for the other algorithms, the

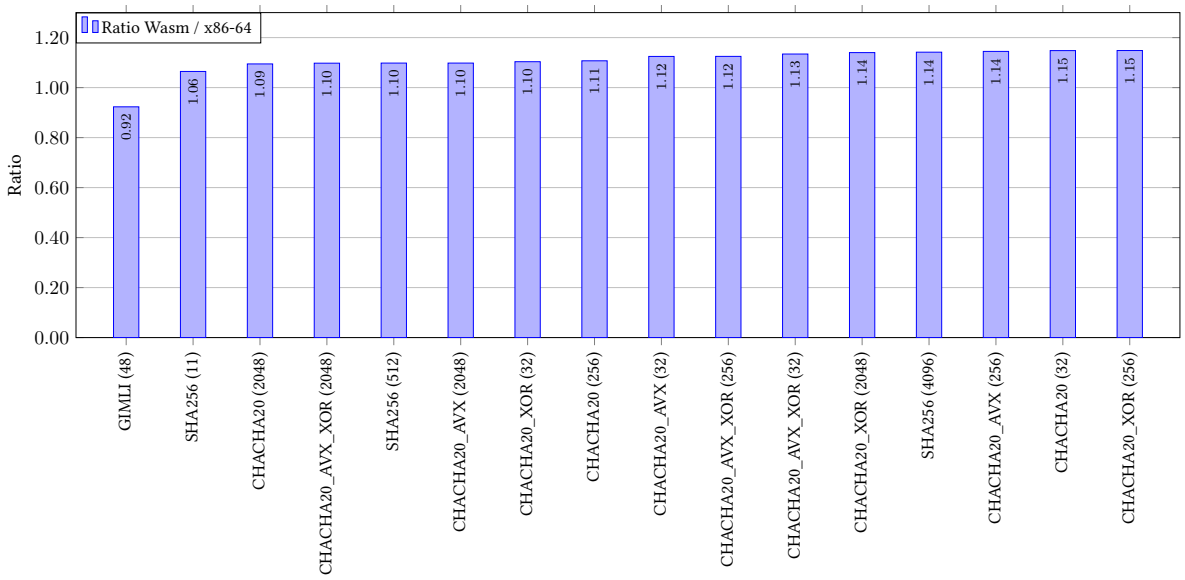

Figure 8: Ratio of execution time on the optimized versions between Wasm and x86-64

x86-64 backend is more efficient with about a 14% to 24% overhead for the minimal versions compared to about 6% to 15% for the optimized versions. The times obtained for the minimal versions are quite acceptable for programs originally designed for x86-64. This allows for rapid prototyping with Wasm from existing Jasmin programs for x86-64 with realistic execution times. Nonetheless, the optimized versions highlight a global and logical improvement in execution times. This gain stems directly from the specialization of the programs to the WebAssembly architecture, while also possibly resulting from their very structure. It is possible that with this specialization, the obtained programs look more like the examples used during the development of `wasm-opt` or the JIT, which allows for better optimization of the generated Wasm programs. This hypothesis remains to be nuanced, however, as the performance gap is not sufficient to exclude the effect of code specialization alone, and at the same time, the programs generated by Jasmin may generally not be in the expected format.

Let us now look at the impact of V8 and SpiderMonkey on performance. We will only present the results on the optimized versions since the trends are relatively the same as with the minimal versions and because Jasmin's philosophy is to have a version of each algorithm specialized to the targeted architecture. It is also possible that the measurements are more indicative when using more realistic WebAssembly programs. Figure 9 presents the results of the benchmark by comparing the ratio between the best execution time obtained with V8 and the best execution time obtained with SpiderMonkey.

V8 clearly outperforms SpiderMonkey on non-vectorized algorithms with ratios ranging from 0.65 to 0.87, while SpiderMonkey proves slightly superior on vectorized instructions with ratios ranging from 1.01 to 1.07. There are potentially two explanations for this. The first would be that V8 is much more performant than SpiderMonkey but that the techniques used by SpiderMonkey (different from those used by V8) work particularly well with vectorized instructions. The other would then be to think that V8's development has not been heavily focused on vectorized instructions and thus that V8 might be lagging behind on these compared to other instructions. Indeed, vectorized instructions were initially not present at the beginning of Wasm, and became available as an extension (which today is an integral part of the standard). What emerges from this comparison between V8 and SpiderMonkey is that to obtain performance closest to that of x86-64, one must use V8 when the

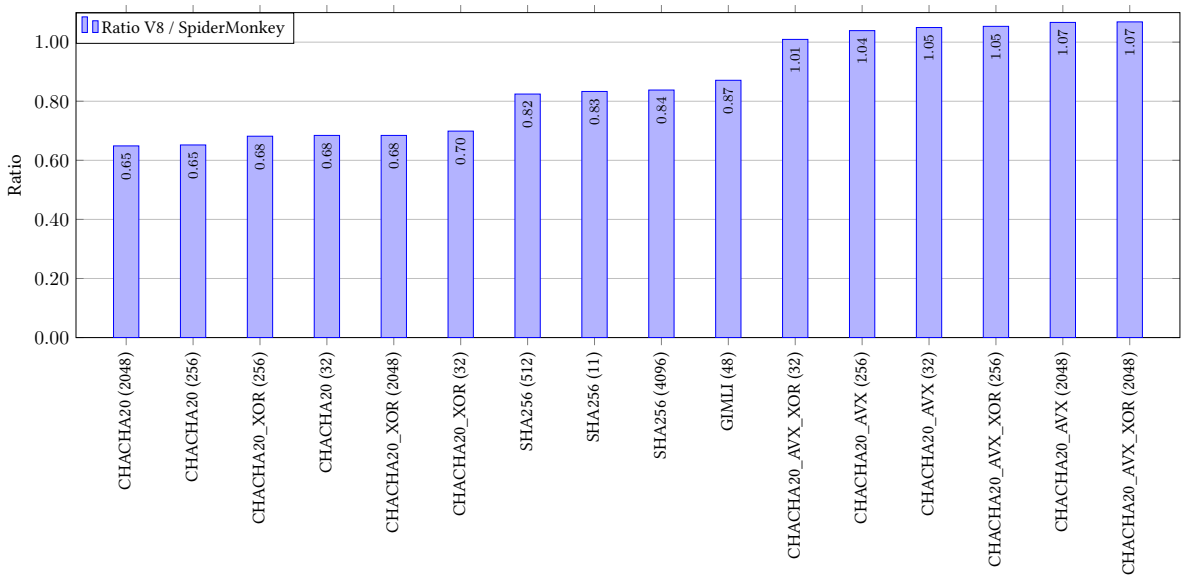

Figure 9: Ratio of execution time on the optimized versions between V8 and SpiderMonkey

program is not based on vectorized instructions and SpiderMonkey when the program is based on vectorized instructions. In this sense, V8 and SpiderMonkey seem to be complementary.

We can conclude from this benchmark that the obtained WebAssembly code is very efficient with performance similar to the x86-64 backend (only 6% to 15% overhead). This means that this backend can be used in practice to execute efficient and secure cryptographic algorithms, notably on the Web.

4.2.2 Interfacing Wasm with Other Languages

After having tested the performance of the produced WebAssembly code on cryptographic algorithms by comparing it to that of x86-64 code, we decided to broaden the treated application domain, notably to replace pieces of program written in JavaScript. For example, real-time image processing in the browser is very costly, and JavaScript struggles to keep up the pace in some cases.

We therefore focused on image processing algorithms. The first applies a blurred reflection to the image, while the second implements a kernel matrix convolution product to accentuate edges or sharpness.

Our approach consists of taking the JavaScript code and rewriting it in Jasmin's syntax, following the logic of the initial code as closely as possible without applying the slightest optimization. Next, we write a second version of this algorithm in Jasmin that attempts to take advantage of the vectorized operations present in Jasmin and Wasm.

For the various upcoming results, we only measured the time elapsed to compute the filter and not the part of the program retrieving the different image data common to JavaScript and WebAssembly. For the algorithm performing a reflection, observe in Figure 10 that the reference version (i.e., without vectorization, without optimization) written in Jasmin is already 2.09 times more efficient than the one written in pure JavaScript. This might seem surprising since two copies are performed to pass the image data to the Wasm part and to subsequently retrieve the result; two copies that are absent in the pure JavaScript version. Nevertheless, the time saved is large enough to amortize the cost of the copies thanks to WebAssembly's more direct semantics, which notably allows the JIT to better

optimize the Wasm code compared to the JavaScript code. Vectorization improves performance to reach a ratio of 2.80. For the algorithm based on kernels, we decided to calculate both transformations for each image. Figure 10 highlights that the reference version is already 77 times more efficient than the pure JavaScript version, still while paying for the two additional copies. By adding vectorization, an impressive ratio of 212 is reached.

Algorithm	Reference	Vectorized
Reflection	2.09	2.80
Kernels	77.04	212.04

Figure 10: Ratio between JavaScript and WebAssembly execution times for image processing

These results are partly explained by the fact that the reflection algorithm performs simpler computations and in a much less intensive manner, notably with an average computation time per image of 1.21ms for JavaScript versus 0.43ms for WebAssembly (vectorized version). By comparison, computing the two transformations using a kernel is much more costly with an average computation time per image of 682.68ms for JavaScript versus 3.22ms for Wasm (vectorized version). JavaScript's semantics mandate many indirections and tests in the implementations of interpreters or JIT, unlike WebAssembly's which is much simpler and more direct (notably due to its strongly typed language), thereby offering far more optimization prospects. This confirms the relatively well-known fact in the literature, namely that Wasm is an ideal candidate to replace intensive computations performed in JavaScript.

Let us explain why adding *intrinsics* is important to achieve optimal performance. Figure 11 presents the Jasmin implementation of the vectorized minimum operation without using an *intrinsic*, along with the generated WebAssembly code. The `min_4s32` function, for each 32-bit integer of `a` and `b`, retrieves the minimum and stores it in the result vector. In a realistic Jasmin program, the `min` function would be unrolled into the body of `min_4s32`, which would itself be unrolled into the rest of the code. Despite the unrolling, which makes the generated code more efficient, the large number of instructions used to compute the minimum significantly slows down the program. Indeed, it is the use of the `#MIN_4s32` *intrinsic*, generating the corresponding native Wasm instruction, that allows obtaining the ratio of 212 versus approximately 90 without it.

```
fn min(reg u32 a b) -> reg u32 {
  reg u32 res;
  if (a <32s b) {
    res = a;
  } else {
    res = b;
  }
  return res;
}

export fn min_4s32(reg u128 a b) -> reg u128 {
  reg u128 res;
  reg u32 va, vb, vres;
  inline int i;
  for i = 0 to 4 {
    va = #EXTRACT_LANE_4s32(i, a);
    vb = #EXTRACT_LANE_4s32(i, b);
    vres = min(va, vb);
    res = #REPLACE_LANE_4s32(i, res, vres);
  }
  return res;
}
```

```
(module $utils
  (func $min (param $a.241 i32) (param $b.242 i32) (result i32)
    (local $res.316 i32)
    (if (i32.lt_s (local.get $a.241) (local.get $b.242))
      (then (local.set $res.316 (local.get $a.241)))
      (else (local.set $res.316 (local.get $b.242))))
    (return (local.get $res.316))
  )
  (func $min_4s32 (param $a.231 v128) (param $b.232 v128) (result v128)
    (local $res.233 v128) ;; ...
    (local $va.320 i32) (local $vb.321 i32) (local $vres.322 i32) ;; ...
    (local.set $va.320 (i32x4.extract_lane 0 (local.get $a.231)))
    (local.set $vb.321 (i32x4.extract_lane 0 (local.get $b.232)))
    (local.set $vres.322
      (call $min (local.get $va.320) (local.get $vb.321)))
    (local.set $res.323
      (i32x4.replace_lane 0 (local.get $res.233) (local.get $vres.322)))
    ;; idem for i = 1, 2 and 3
    (return (local.get $res.335))
  )
  (export "min_4s32" (func $min_4s32))
)
```

Figure 11: A vectorized minimum Jasmin implementation without using an *intrinsic* along with the generated Wasm code

5 Conclusion

5.1 Summary

We first developed in OCaml a complete backend for WebAssembly. Relying on a Wasm architecture formalised in Rocq, we were able to leverage the existing frontend. This backend incorporates specific compilation passes that adapt the source program prior to its final translation. All instructions present in Jasmin are supported, as well as several WebAssembly-specific instructions such as SIMD instructions.

We also extended the Jasmin language to introduce a new type: references. This new type made it possible to utilize the new WasmGC extension, enabling generated code to interface with higher-level garbage-collected languages such as Dart, Haskell, OCaml, Ruby, or Scheme.

To consolidate confidence in the correctness of the new backend, a test suite was established. It consists of differential tests for Jasmin instructions common to all architectures, as well as unit tests for WebAssembly-specific instructions.

The comparative evaluation of the x86-64 and Wasm backends reveals encouraging results: the measured overhead is only 10% to 15% on equivalent Jasmin cryptographic programs.

Furthermore, we wrote Jasmin programs specialized for the WebAssembly backend to perform image processing. To optimize these developments, we integrated additional specific Wasm instructions into Jasmin. The benchmarks confirm that in compute-intensive scenarios, their use provides a significant efficiency gain compared to JavaScript.

5.2 Limitations

The first limitation lies in our handling of memory within the resulting WebAssembly programs. Ideally, Jasmin's source memory and external memory should be strictly separated. This is not currently the case: the intermediate representation of the code does not allow distinguishing the origin of memory accesses. Overcoming this limitation would require modifying the compiler to ensure the traceability of these accesses.

A second, more restrictive limitation concerns the lack of a functional correctness proof for the backend. Its current implementation in OCaml stands in the way of formal verification, making it necessary to reimplement it in Rocq.

5.3 Future Work

A primary objective would be to reimplement the backend directly in Rocq to prove that compilation to WebAssembly preserves the functional correctness of Jasmin programs. This work will build upon a formal semantics of WebAssembly that incorporates SIMD instructions and the WasmGC extension.

Another direction would be to prove that the entire compilation pipeline preserves the cryptographic constant-time property. This would involve targeting the constant-time extension of Wasm as a compilation target.

It is also worth investigating the impact of JIT compilation on the constant-time property. Indeed, this compilation mode is liable to compromise resistance to side-channel attacks, much like the optimizations performed by traditional compilers. Since disabling it is out of the question for performance reasons, the challenge will be to devise a method that forces the JIT compiler to preserve the

constant-time property.

5.4 Acknowledgments

I would like to thank the entire lab team for their warm welcome, hospitality, and kindness. I particularly thank Jean-Christophe Léchenet for his invaluable help and expertise on Jasmin. Finally, I am deeply grateful to Benjamin Grégoire and Manuel Serrano for their enthusiasm regarding the work accomplished, their excellent supervision, and their guidance throughout this enriching research internship.

References

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. *CoRR*, abs/1904.04606, 2019. URL http://arxiv.org/abs/1904.04606.

José Bacelar Almeida, Santiago Arranz Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, and Pierre-Yves Strub. Formally verifying kyber: Episode v: Machine-checked ind-cca security and correctness of ml-kem in easycrypt. In *Advances in Cryptology – CRYPTO 2024: 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2024, Proceedings, Part II*, page 384–421, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-3-031-68378-7. doi: 10.1007/978-3-031-68379-4_12. URL https://doi.org/10.1007/978-3-031-68379-4_12.

Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. High-assurance cryptography in the spectre era. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1884–1901, 2021. doi: 10.1109/SP40001.2021.00046.

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 *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. doi: 10.1145/3062341.3062363. URL https://doi.org/10.1145/3062341.3062363.

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 17 Edition.* Oracle America, Inc., 2021. URL https://docs.oracle.com/javase/specs/jvms/.

Manuel Serrano and Robert Bruce Findler. A snapshot of the performance of wasm backends for managed languages. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR '25, page 93–106, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400721496. doi: 10.1145/3759426.3760983. URL https://doi.org/10.1145/3759426.3760983.

Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. Ct-wasm: type-driven secure cryptography for the web ecosystem. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi: 10.1145/3290390. URL https://doi.org/10.1145/3290390.

Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. Bringing the webassembly standard up to speed with spectec. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi: 10.1145/3656440. URL https://doi.org/10.1145/3656440.