Rapport de stage
De Jasmin à Wasm

Hugo Barreiro M2 MPRI
Supervisé par Benjamin Grégoire et Manuel Serrano
Centre Inria d'Université Côte d'Azur, équipe SPLiTS

Mars 2026 — Juillet 2026

Contexte général

Jasmin est un environnement dédié à la cryptographie performante de confiance. Ses implémentations sont pensées pour être efficaces, sûres, correctes et sécurisées. Le langage de programmation Jasmin associe des constructions de haut et de bas niveau, permettant au développeur d'anticiper le code assembleur généré. Afin de raisonner de manière rigoureuse sur le comportement des programmes, sa sémantique est définie formellement, ce qui permet de prouver des propriétés sur ces programmes.

WebAssembly est une machine virtuelle récente dont l'objectif est de fournir une plateforme d'exécution universelle. Malgré son nom, elle a été conçue comme un langage assembleur polyvalent, à l'instar de la JVM, plutôt que spécifiquement pour le Web. Un des points forts de Wasm est son modèle d'exécution sûr, ce qui en fait une cible prometteuse pour toutes les applications critiques, et en particulier les calculs cryptographiques.

Problématique

L'objectif de ce stage est d'implémenter dans l'environnement Jasmin un nouveau générateur de code WebAssembly. Bien que d'autres langages proposent déjà cette cible de compilation, l'intégrer à Jasmin permettrait de profiter de ses atouts comme l'accès à la vérification formelle des programmes, l'efficacité du code généré ou encore des abstractions de haut niveau proposées dans un langage bas niveau.

L'utilisation d'un *backend* WebAssembly permettrait d'implémenter des programmes sûrs et efficaces pour le Web, notamment des algorithmes cryptographiques. Ainsi, il serait possible de tirer avantage de Jasmin pour vérifier toutes les propriétés nécessaires grâce à l'extraction vers EasyCrypt, un assistant de preuves spécialisé pour les preuves cryptographiques.

Jasmin étant un langage généraliste et Wasm une plateforme universelle, rien n'empêche de les utiliser dans d'autres contextes. Par exemple, il est envisageable d'écrire des algorithmes de traitement d'image en Jasmin de manière plus agréable, voire plus efficace qu'en JavaScript pour des applications web. D'autre part, les programmes cryptographiques WebAssembly obtenus pourraient s'interfacer avec d'autres langages, par exemple OCaml via Wasm-of-OCaml. Cette approche permet d'externaliser les sections critiques en Jasmin pour les intégrer à d'autres écosystèmes.

Contribution

Nous avons développé, en OCaml, un *backend* complet pour générer du code Wasm. Nous y avons également intégré des primitives spécifiques, notamment des instructions vectorisées nécessaires à l'adaptation des programmes Jasmin initialement conçus pour l'architecture x86-64. Pour ce faire, nous avons dû définir formellement, en Rocq, une version minimale de l'architecture WebAssembly afin de pouvoir connecter le *frontend* du compilateur Jasmin au nouveau *backend*. Enfin, nous avons étendu le langage Jasmin avec l'introduction du type référence afin de pouvoir utiliser l'extension WasmGC avec les programmes générés.

Le code généré s'avère particulièrement efficace : les différents *benchmarks*, effectués sur des programmes cryptographiques utilisables en pratique, montrent des performances comparables à celles du *backend* x86-64. Pour des codes sources identiques, les deux cibles sont similaires. Les programmes initialement optimisés pour x86-64 ne subissent qu'un surcoût d'environ 14% à 24% après des modifications minimales de compatibilité. L'écart tombe entre environ 6% et 15% seulement après une spécialisation naïve pour WebAssembly.

Nous avons, par ailleurs, écrit des programmes Jasmin spécialisés pour le *backend* Wasm pour réaliser du traitement d'images. Afin d'obtenir les performances maximales de ces programmes, nous avons eu besoin d'ajouter dans Jasmin d'autres instructions moins courantes et spécifiques à WebAssembly. Ces mesures mettent en évidence un gain de performance significatif par rapport à JavaScript lorsque les calculs intensifs sont délégués au code Wasm produit par Jasmin.

Arguments de validité

Ce *backend* étant écrit en OCaml et non en Rocq, sa correction fonctionnelle ne peut pas encore être formellement prouvée. Néanmoins, nous avons mis en place une suite de tests, incluant notamment des tests unitaires pour les instructions spécifiques à l'architecture WebAssembly. Pour les instructions Jasmin, leur traduction vers Wasm est validée par des tests différentiels avec l'architecture x86-64, dont la correction fonctionnelle est formellement prouvée.

Synthèse et perspectives

Nous avons implémenté un *backend* en OCaml pour générer du code Wasm. Il supporte l'intégralité du jeu d'instructions Jasmin, enrichi de plusieurs primitives propres à WebAssembly. Les *benchmarks* montrent des performances satisfaisantes, notamment seulement environ 10% à 15% de surcoût par rapport au code x86-64. Des mesures complémentaires indiquent que le code Jasmin spécialisé pour Wasm constitue une alternative prometteuse pour remplacer les calculs intensifs écrits en JavaScript.

Plusieurs travaux peuvent être encore menés. À court terme, la réimplémentation du *backend* en Rocq permettrait de certifier formellement sa correction. Une autre piste consisterait à exploiter l'extension *constant-time* de WebAssembly afin de préserver *constant-time* sur le code généré, avant d'étudier l'impact de la compilation JIT sur cette propriété.

Table des matières

1 Introduction

L'objectif principal de ce stage est d'ajouter, dans le compilateur Jasmin, un nouveau *backend* générant du code WebAssembly.

1.1 Motivation

Implémenter des algorithmes cryptographiques présente des défis majeurs. En plus de devoir être particulièrement efficaces pour ne pas ralentir toute application critique, l'implémentation se doit également d'être correcte afin de ne pas compromettre des données sensibles. Garantir ces deux aspects est compliqué puisque l'optimisation d'un programme complexifie souvent son analyse. Par exemple, les applications web requièrent une grande efficacité mais leur implémentation est difficilement prouvable, notamment si elles sont écrites avec des langages comme JavaScript.

C'est ici qu'intervient l'idée d'associer Jasmin avec WebAssembly. Cette combinaison permettrait d'apporter la performance du code Wasm avec les certitudes de Jasmin, en particulier la possibilité de prouver le bon comportement du programme WebAssembly obtenu. Ainsi, ce code pourrait être utilisé sur le Web afin d'allier efficacité et sûreté.

Ce nouveau *backend* faciliterait également l'interfaçage du code avec des langages de haut niveau comme OCaml. En effet, Jasmin offre un cadre plus généraliste que la cryptographie avec un contrôle plus fin sur le code assembleur final. Cela offrirait l'opportunité d'apporter un support supplémentaire dans des langages où les questions bas niveau sont habituellement abstraites.

Dans ce rapport, nous allons présenter comment implémenter un nouveau *backend* dans Jasmin pour générer du code Wasm efficace qui pourra être utilisé en pratique. Nous tirerons avantage des structures déjà mises en place par cette plateforme.

1.2 Intuition

L'objectif est de s'appuyer sur les structures existantes de la plateforme Jasmin plutôt que de les réimplémenter intégralement. En particulier, la plupart des passes de compilation présentes dans le *frontend* peuvent être reprises. Néanmoins, cela n'est pas vraiment possible pour le *backend*. En effet, un programme WebAssembly a une structure bien différente des autres langages assembleurs supportés jusqu'à présent par Jasmin, comme x86-64, RISC-V ou Arm M4. Wasm est une machine à pile avec une mémoire linéaire dont le code, basé sur des opérateurs de flux de contrôle structurés, est compilé dynamiquement. Alors que les autres architectures sont des machines à registres avec une mémoire physique dont le code, basé sur des étiquettes et des `goto`, est compilé statiquement.

Il faut d'abord définir une architecture pour WebAssembly en Rocq, qui par ailleurs peut être plus ou moins minimale dans un premier temps. Cela consiste à déclarer les informations essentielles sur Wasm comme la taille du pointeur de données, les types de registres, les types de drapeaux, la convention d'appel, certaines instructions, etc. Dès lors, il devient possible de réutiliser le *frontend* déjà présent afin d'utiliser le programme Jasmin transformé comme entrée du futur *backend*.

Dans le *backend* lui-même, des passes de compilation plus spécifiques à l'architecture WebAssembly peuvent être effectuées afin de transformer davantage le programme Jasmin. Une fois suffisamment préparé, le programme Jasmin peut être traduit en Wasm. Cette tâche consiste à identifier les instructions qui peuvent être traduites de manière suffisamment prédictible afin de préserver la philosophie de Jasmin qui garantit une correspondance quasi directe entre les instructions du programme source et le code assembleur généré.

Pour être convaincu de l'efficacité des programmes WebAssembly obtenus, leurs performances sont comparées à celles des autres *backends* pour les mêmes programmes cryptographiques réalistes écrits en Jasmin. Une suite de tests différentiels est également mise en œuvre avec un autre *backend*, dont la correction est formellement prouvée, pour renforcer la confiance dans la correction du *backend* Wasm.

2 Contexte

Cette section présente les défis liés à l'implémentation d'algorithmes cryptographiques, puis introduit Jasmin et WebAssembly en détaillant leurs spécificités, définitions et possibles applications.

2.1 Implémentation d'algorithmes cryptographiques

La cryptographie est un domaine critique pour plusieurs raisons. Tout d'abord, les implémentations de ces algorithmes sont massivement utilisées, tant en local qu'en ligne. Leur efficacité est donc critique pour ne pas ralentir les applications. Ensuite, elles sont à la base des protocoles de sécurité, il est donc crucial d'en garantir la correction puisqu'un simple bogue compromettrait l'ensemble du système. Enfin, bien que les algorithmes soient cryptographiquement sûrs, leurs implémentations ne le sont pas pour autant puisqu'elles sont vulnérables aux attaques par canaux cachés. Afin de pouvoir y résister, les contraintes imposées par le matériel doivent être prises en compte.

Dès lors, la question de leur implémentation pratique se pose. Utiliser des langages structurés pose problème car les optimisations réalisées par les compilateurs cassent la résistance aux attaques par canaux cachés. En effet, il est courant que les développeurs utilisent des motifs de masquage afin d'éviter que les données secrètes dépendent des branchements ou des accès mémoire du programme : c'est ce que désigne la propriété *constant-time*. Malheureusement, les compilateurs optimisants parviennent à identifier ces motifs et à les remplacer par des structures classiques, annihilant ainsi la protection apportée. Il est donc courant que les développeurs finissent par implémenter les parties critiques des algorithmes directement en assembleur afin d'avoir un contrôle total sur le code final. Néanmoins, l'écriture directe en assembleur présente des inconvénients majeurs : cette tâche fastidieuse complexifie la maintenance. De plus, leur absence inhérente de structuration et d'abstractions rend le raisonnement sur le code particulièrement difficile. C'est donc pour répondre à ces enjeux posés par la cryptographie qu'un nouveau langage a été conçu : Jasmin.

2.2 Jasmin

Jasmin [Almeida et al., 2019, Barthe et al., 2021, Almeida et al., 2024] est un environnement dédié à la cryptographie performante de confiance. Ses implémentations doivent être efficaces, sûres, correctes et sécurisées. Jasmin est un langage bas niveau offrant un contrôle fin sur l'assembleur généré permettant d'obtenir du code efficace préservant les propriétés de sécurité. De plus, il propose des constructions de haut niveau comme des variables, des fonctions, des tableaux ou encore des boucles assurant une bonne structuration des programmes. Enfin, sa sémantique formelle simple et claire, associée à ces abstractions, facilite le raisonnement sur les programmes, contrairement aux langages bas niveau traditionnels. Ainsi, le compilateur Jasmin produit du code assembleur prévisible en plus de garantir que l'utilisation d'abstractions de haut niveau n'a aucun coût à l'exécution.

Jasmin étant développé et vérifié en Rocq, plusieurs propriétés clés du compilateur sont formellement garanties. Il est notamment démontré que la sémantique du programme source est préservée par le code généré. Par ailleurs, la correction fonctionnelle de chaque programme Jasmin est prouvable grâce à un mécanisme d'extraction vers EasyCrypt, un assistant de preuve spécialisé pour les preuves cryptographiques. Ainsi, toute propriété établie sur le code source est intrinsèquement conservée lors

de la compilation. Enfin, le système de types de Jasmin garantit que si le programme est bien typé alors il est *constant-time*, c'est-à-dire qu'il résiste aux attaques par canaux cachés.

Bien que propice aux implémentations cryptographiques, Jasmin reste avant tout un langage généraliste, ce qui permet de l'utiliser dans d'autres contextes notamment en l'interfaçant avec d'autres langages grâce à ses différents *backends*, le tout en profitant de ses avantages comme l'utilisation d'instructions assembleur précises dans un langage structuré ou encore la facilité à prouver des propriétés sur les programmes générés.

2.3 WebAssembly

WebAssembly [Haas et al., 2017, Youn et al., 2024], aussi appelé Wasm, est une machine virtuelle récente dont l'objectif est de fournir une plateforme d'exécution universelle, avec dès à présent des compilations depuis C, OCaml et Rust. Malgré son nom, WebAssembly ne se limite pas au Web : il s'agit d'une machine virtuelle avec un langage assembleur polyvalent, à l'instar de la JVM [Lindholm et al., 2021].

Le langage proposé est fortement typé, sûr vis-à-vis de la mémoire, s'exécute en mode bac à sable, et ne supporte que les opérateurs de flux de contrôle structurés comme les boucles ou les fonctions. Wasm se démarque des autres machines virtuelles de par son agnosticisme vis-à-vis du langage source et sa proximité avec les plateformes matérielles en ne nécessitant qu'un système d'exécution minimal, ce qui en fait une machine virtuelle portable et facilement interfaçable avec notamment des utilisations dans le navigateur.

La sécurité de son modèle d'exécution constitue un atout majeur : le typage strict exclut les comportements indéfinis, l'isolation en bac à sable empêche les accès illégitimes à la mémoire hôte (immunisant le système contre les dépassements de tampon), et toute interaction avec le système d'exploitation requiert une autorisation explicite. Cela en fait une cible prometteuse pour toutes les applications critiques, et en particulier les calculs cryptographiques. De plus, l'extension SIMD introduit des instructions vectorisées à 128 bits, ce qui renforce sa compatibilité avec la cryptographie performante où le parallélisme des opérations est essentiel.

WebAssembly possède bien d'autres extensions, telles que WasmGC qui permet de l'interfacer de manière efficace [Serrano and Findler, 2025] avec des langages de haut niveau comme Dart, Haskell, OCaml, Ruby ou Scheme grâce à l'introduction d'un ramasse-miettes. WasmGC a d'ailleurs été récemment ajouté à la version 3.0 de la spécification de WebAssembly. Il existe également CT-Wasm [Watt et al., 2019] qui permet d'assurer que les programmes WebAssembly soient bien *constant-time*.

3 Le compilateur Jasmin

Cette section explique l'objectif du compilateur Jasmin en plus de détailler son langage ainsi que son architecture.

3.1 Objectif

L'objectif principal du compilateur Jasmin est d'offrir un langage garantissant un contrôle strict sur l'assembleur généré tout en proposant des constructions de haut niveau facilitant l'implémentation et la preuve formelle par rapport aux différents langages assembleur. Pour ce faire, le langage permet, entre autres, de choisir ses instructions (instruction Jasmin ou instruction assembleur) ou encore le type de mémoire utilisé (registre ou pile).

Afin de garantir cette prédictibilité, le compilateur s'interdit toute transformation lourde du code. Ainsi, une instruction Jasmin doit moralement correspondre à une instruction assembleur. Il n'est pas envisageable d'émuler certains comportements si cela dégrade l'efficacité du programme (en temps ou en espace) par rapport à l'usage des instructions natives.

Il existe une différence fondamentale entre les programmes Jasmin valides (ceux qui possèdent une sémantique) et les programmes Jasmin qui peuvent être compilés. En effet, la sémantique étant indépendante des architectures cibles, certaines d'entre elles peuvent ne pas disposer d'instructions capables d'exprimer nativement certaines constructions valides. Dans ce cas, le compilateur préfère échouer. Il est ainsi courant que des programmes valides ne compilent pas pour respecter la prédictabilité et le contrôle strict sur l'assembleur généré. La philosophie de Jasmin privilégie la perte de portabilité contre l'efficacité pour rivaliser avec du code natif. Il est donc préconisé de maintenir, pour chaque algorithme, une version spécialisée pour chaque architecture.

3.2 Langage

Jasmin possède plusieurs types d'entiers. D'une part, le type `int` représente les entiers de précision infinie, dont la valeur doit obligatoirement être connue statiquement. D'autre part, les types d'entiers sur 8, 16, 32, 64, 128 et 256 bits acceptent des expressions arbitraires. Selon l'architecture cible, certaines tailles d'entiers ne sont pas disponibles en raison des limitations apportées par les différentes plateformes. Il est possible d'effectuer des opérations de conversion entre les différents types d'entiers. Pour transtyper vers un entier plus précis, la conversion doit être explicite afin de savoir si l'utilisateur veut réaliser une extension de signe ou bien une extension de zéros, deux opérations pouvant donner deux résultats différents. En revanche, pour transtyper vers un entier moins précis, la conversion est implicite puisqu'il existe exactement une façon de la réaliser. Les entiers supportent les opérations arithmétiques, signées et non-signées, ainsi que les opérations bit à bit et logiques usuelles.

Le langage propose également des tableaux dont la taille doit être connue statiquement. Outre les opérations classiques de lecture et d'écriture, Jasmin permet de choisir explicitement leur emplacement en mémoire (registre ou pile) via les mots-clefs *reg* et *stack*. Deux autres types, *reg ptr* et *stack ptr*, agissent moralement comme des pointeurs vers des tableaux stockés respectivement dans des registres ou sur la pile. Stocker des tableaux en pile peut s'avérer utile pour soulager la pression sur l'utilisation des registres afin de faciliter ou bien rendre possible leur allocation.

Les programmes Jasmin sont structurés en fonctions non récursives. Ces dernières peuvent être marquées avec des mots-clefs spécifiques. Par exemple, les fonctions `inline` indiquent que le corps de ces fonctions sera déroulé à l'endroit de l'appel. Les fonctions `export` quant à elles indiquent que ces fonctions pourront être appelées par l'environnement extérieur. En effet, Jasmin est prévu pour être utilisé par des programmes d'autres langages. C'est notamment le cas de C qui peut utiliser le code x86-64 produit par Jasmin. Par ailleurs, pour aider à la structuration des programmes, le langage propose des boucles `for` et des boucles `while`. Les bornes des boucles `for` doivent être connues statiquement puisque ces dernières seront entièrement déroulées. À l'inverse, les boucles `while` permettent d'exprimer des boucles de longueurs arbitraires, sans déroulement.

En plus des instructions usuelles sur les entiers et tableaux, Jasmin propose d'utiliser directement des instructions assembleur provenant des différentes architectures, désignées sous le nom d'*intrinsics*. La figure 1 présente une utilisation d'*intrinsics* x86-64 pour effectuer l'addition exacte à 128 bits efficacement. Les *intrinsics* permettent d'exploiter l'addition avec retenue pour utiliser seulement deux instructions, alors que sans, l'utilisateur est obligé d'effectuer une comparaison pour connaître la valeur de la retenue, rendant le code produit moins performant. Si elles permettent d'exprimer

des calculs optimisés de façon très efficace, elles rendent en contrepartie le programme dépendant de l'architecture cible, empêchant sa compilation vers les autres plateformes. Cette fonctionnalité reflète l'essence même de Jasmin, à savoir la spécialisation à l'extrême au détriment de la portabilité.

```
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;
}
```

FIG. 1 : Deux implémentations Jasmin d'une addition exacte à 128 bits, une avec *intrinsics* x86-64, l'autre sans

3.3 Architecture

Les différentes passes de compilation de Jasmin sont illustrées par la figure 2. Elles sont toutes formellement vérifiées, soit en étant directement écrites et prouvées en Rocq, soit par un vérificateur prouvé en Rocq qui valide l'exactitude du résultat de la passe. Le *frontend* regroupe les passes allant du langage Jasmin au langage intermédiaire Stack, tandis que le *backend* rassemble les passes dépendantes de l'architecture cible, allant de Stack à l'ASM.

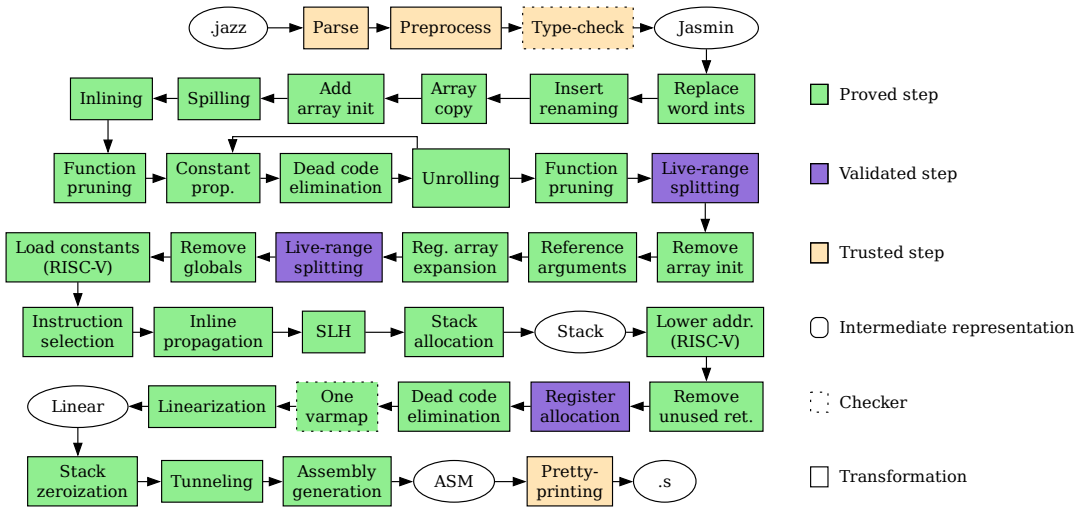

FIG. 2 : Les différentes passes de compilation de Jasmin

Le *frontend* applique au programme Jasmin typé les transformations nécessaires à la génération d'assembleur. Cela commence par le remplacement des *word ints* (un type d'entier machine traité comme des `int` dans la sémantique) en entiers machine, le renommage de certaines variables des fonctions pour se conformer aux conventions d'appel, le remplacement des pseudo-instructions `#copy`, `#spill` et `#unspill` par leurs équivalents physiques (copies réelles, mise en pile ou registre), le déroulement des fonctions marquées par *inline*, la suppression des fonctions inutilisées (comme les fonctions *inline* qui ne sont pas *export*). Ensuite, une phase de simplification répète autant de fois que possible la propagation des constantes, l'élimination du code mort et le déroulement des boucles `for`. Les fonctions devenues inutilisées sont supprimées. Le compilateur continue par une sorte de mise en forme SSA (*Static Single Assignment*) pour réduire la durée de vie des variables, l'insertion

des variables *ptr* pour les arguments et valeurs renvoyées des fonctions, la transformation de chaque case des tableaux en des variables uniques et indépendantes, puis par la mise en forme SSA des nouvelles variables. Les instructions Jasmin sont alors remplacées par des instructions assembleur selon l'architecture cible. Enfin, les variables marquées *inline* sont propagées avant de passer à l'allocation de la pile. Cette étape cruciale consiste à réserver des zones de la pile pour stocker les variables globales et les variables locales marquées *stack* (dont les tableaux) de chaque fonction.

L'analyse de ce processus met en évidence l'efficacité de la propagation des constantes, maximisée par l'usage des variables `int`, des boucles `for` statiques et des fonctions *inline*. Ces passes simplifient ou éliminent de nombreuses instructions intermédiaires, optimisant ainsi le programme source. Notons que les tableaux sont entièrement déroulés : chaque case devient une variable stockée dans des registres ou sur la pile.

Le *backend* finalise la compilation vers l'assembleur. Il supprime d'abord les valeurs de retour inutilisées des fonctions *export*, puis procède à l'allocation des registres. Lors de cette étape critique, toutes les variables *reg* (y compris celles issues des tableaux déroulés) sont associées à des registres. Cette phase échoue si l'architecture cible ne dispose pas d'assez de registres pour satisfaire les contraintes du programme. En cas de succès, le compilateur élimine le code mort résiduel, explicite la gestion de la pile et des arguments, et traduit le contrôle de flot structuré en étiquettes et instructions `goto`. Ces dernières sont finalement aplanies avant la génération du code assembleur final.

Ainsi, bien qu'un programme soit correctement typé, la compilation peut échouer lors de la sélection d'instructions (si une primitive n'existe pas sur l'architecture cible) ou lors de l'allocation de registres (en cas de saturation).

4 Stage

Cette section présente le travail effectué pendant le stage et les principaux résultats obtenus. L'archive du code est disponible sur GitHub.

4.1 Le *backend* WebAssembly

Cette section détaille l'implémentation du *backend* Wasm. Nous présentons d'abord la définition minimale de l'architecture WebAssembly en Rocq, puis la traduction du code Jasmin vers Wasm, implémentée en OCaml.

4.1.1 Définition minimale de l'architecture

Le nouveau *backend* WebAssembly s'insère juste après l'allocation de la pile. Les passes du *frontend* profitent à toutes les architectures en optimisant le code source et gérant les tableaux, notamment leur mise en pile. Néanmoins, les passes réalisées par les autres *backend* ne présentent aucun intérêt dans le cas de Wasm. Contrairement aux autres architectures, WebAssembly est une machine à pile s'appuyant sur des variables globales ou locales et un flux de contrôle structuré. L'allocation de registres et la linéarisation sont donc inutiles. Nous définissons ainsi une architecture minimale permettant d'utiliser le *frontend* pour insérer le *backend* Wasm juste après l'allocation de la pile.

Dans cette architecture, il n'y a pas de registres, uniquement des variables locales ou globales. La mémoire est composée d'un ou de plusieurs tableaux d'octets. Les appels de fonctions sont réalisés avec l'instruction `call`. Cela permet de spécifier que le pointeur de données est un entier 32 bits et qu'il n'y a ni registre, ni drapeau, ni convention d'appel explicite.

Cette étape implique également la déclaration des instructions WebAssembly, c'est-à-dire leur nom, validité, type, sémantique et la nature des arguments (expressions arbitraires ou immédiats). Si ces définitions ne sont pas strictement requises pour le *backend* en OCaml, elles s'avèrent essentielles pour sa preuve de correction en Rocq et permettent d'exploiter les *intrinsics* dans les programmes Jasmin.

Le langage Jasmin a pour objectif d'être supporté par toutes les architectures et il peut s'avérer restrictif pour les architectures dotées d'un jeu d'instructions particulièrement riche. Par conséquent, des instructions spécifiques à Wasm doivent être absolument déclarées. Par exemple, les implémentations les plus efficaces des algorithmes cryptographiques se basent sur des instructions vectorisées afin d'effectuer des calculs en parallèle. Jasmin propose ce genre d'instructions, mais certaines ne sont pas disponibles. Dans le cas de WebAssembly, il faut donc déclarer quelques instructions SIMD utiles supplémentaires, c'est-à-dire des instructions vectorisées de 128 bits permettant d'effectuer des calculs arithmétiques en parallèle impossibles à exprimer efficacement avec le langage Jasmin.

Il devient possible d'exploiter le *frontend* du compilateur qui est donc commun à chaque architecture. Par ailleurs, certaines passes de compilation du *frontend* peuvent ne pas être appliquées dans le cas de Wasm, notamment celle qui introduit un renommage de certaines variables pour faciliter l'allocation des registres, qui n'existe pas en WebAssembly.

4.1.2 Traduction de Jasmin à Wasm

Avant de traduire le programme transformé par le *frontend*, il faut ajouter une passe spécifique à l'architecture WebAssembly. La notion de conversion implicite ne pose pas de problème avec les autres architectures puisqu'elles en possèdent également une. Néanmoins, ce n'est pas le cas de Wasm qui est un langage fortement typé, ce qui interdit par exemple l'addition d'un entier 32 bits et d'un entier 64 bits. Un transtypage préalable de l'un des opérandes est indispensable pour obtenir une opération valide, rendant obligatoire l'explicitation de toutes les conversions.

Pour expliciter ces conversions, l'arbre de syntaxe abstraite est parcouru afin de récupérer le type attendu pour chaque argument d'une opération et leur type actuel, puis ajouter une conversion si nécessaire. Par exemple, si l'addition attend deux entiers 32 bits et qu'actuellement l'un d'entre eux est un entier 64 bits, alors il faut ajouter la conversion vers un entier 32 bits. Les conversions successives sont ensuite aplanies. Par exemple, un transtypage de 64 bits vers 32 bits puis 16 bits devient un transtypage direct de 64 bits vers 16 bits.

Une fois cette passe effectuée, le programme Jasmin transformé est alors compilé vers WebAssembly. Il s'agit d'abord d'identifier les constructions non supportées. Jasmin devant garantir la prédictibilité et le contrôle strict du code assembleur produit, le *backend* doit échouer si l'utilisateur déclare des variables de 8, 16 ou 256 bits, types inexistants en Wasm. Représenter ces variables par des entiers 32 bits altérerait cette prédictibilité. Ainsi, toutes les instructions Jasmin qui n'ont pas d'équivalent suffisamment direct en WebAssembly ne peuvent pas être compilées avec ce nouveau *backend*. Par ailleurs, Wasm n'est pas un cas isolé puisqu'il existe des configurations similaires avec les architectures RISC-V et Arm M4.

Les opérateurs de décalage arithmétique illustrent une autre divergence sémantique. En Jasmin, si l'opérande de décalage est trop grand, par exemple 33 pour un opérateur 32 bits, alors le résultat est égal à 0. En WebAssembly, ce n'est pas le cas, l'opérande est évalué modulo 32, donc le résultat sera un décalage de 1. Seuls les décalages conformes à la sémantique Jasmin sont compilables : l'utilisateur

doit insérer explicitement le modulo dans le code source. Ce motif sera reconnu par le *backend* qui le compilera sans calculer le modulo, Wasm l'effectuant implicitement. La figure 3 illustre la compilation de ces constructions.

```
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))
)
```

FIG. 3 : La compilation, de Jasmin à WebAssembly, de certaines opérations de décalage

Il faut également être capable de reconnaître d'autres motifs. Wasm autorise à lire seulement 8 ou 16 bits à une certaine adresse mémoire. Néanmoins, le résultat sera donné par un entier 32 ou 64 bits via une extension de signe ou de zéros. Si l'utilisateur, dans son programme Jasmin, décide de lire 8 ou 16 bits depuis la mémoire et de transtyper le résultat dans un entier 32 ou 64 bits, alors il faut être capable d'identifier cette situation pour pouvoir la compiler vers l'instruction WebAssembly correspondante. Wasm permet également d'écrire seulement 8 ou 16 bits à une certaine adresse mémoire à partir d'un entier 32 ou 64 bits. Une configuration similaire survient lorsque l'utilisateur décide d'écrire seulement 8 ou 16 bits en mémoire en transtypant son entier 32 ou 64 bits. Le *backend* doit donc reconnaître ce motif précis pour le compiler vers l'instruction correspondante. La figure 4 illustre la compilation des différents cas évoqués.

```
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))
)
```

FIG. 4 : La compilation, de Jasmin à WebAssembly, de certains accès mémoire

Les variables *reg*, habituellement stockées dans des registres, sont placées dans des variables locales dans le code généré. En WebAssembly, il n'est pas possible de manipuler directement la pile pour y stocker des valeurs, les variables *stack* sont donc quant à elles stockées dans la mémoire linéaire de Wasm : espace dans lequel la pile est émulée comme décrit dans la figure 5. Les fonctions Jasmin n'étant pas récursives, la taille des blocs d'activations est donc connue statiquement. Ainsi, avec les variables globales, ils sont stockés en mémoire dès l'initialisation du module WebAssembly.

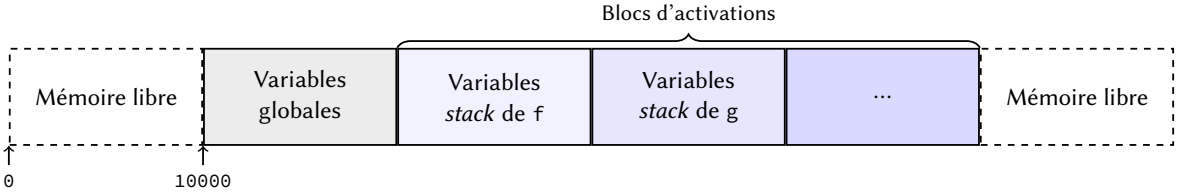

FIG. 5 : Organisation de la mémoire WebAssembly pour les programmes Jasmin

Dans les autres architectures, les variables sont stockées sur la pile. Par construction, lorsque Jasmin est appelé par un autre programme, cette pile est séparée de l'environnement externe permettant

d'avoir une séparation complète entre la mémoire source Jasmin et la mémoire du programme appelant. Or, ce n'est pas le cas ici avec cette gestion de la mémoire en Wasm. En effet, la mémoire utilisée par le programme source peut être modifiée par l'appelant. Il est d'usage que l'utilisateur initialise ou récupère la mémoire du module pour y écrire des informations afin de communiquer avec le programme WebAssembly.

Une première complication se trouve donc dans l'initialisation de cette mémoire puisque l'utilisateur n'a aucun moyen de savoir quelles parties de la mémoire vont être utilisées par le programme Wasm et où il peut l'utiliser de manière sûre. La deuxième complication est qu'elle rend la preuve de correction du *backend* bien plus complexe puisque pour les autres architectures la mémoire source et la mémoire externe étaient forcément séparées. Heureusement, il existe une solution, qui pour l'instant, n'a pas été implémentée. Elle consisterait à marquer les accès mémoire avec un booléen indiquant s'il s'agit d'un accès source ou externe. Cela permettrait d'introduire, dans le programme WebAssembly généré, deux mémoires différentes et donc séparées. Néanmoins, cette solution impose de mettre à jour toutes les preuves du compilateur puisqu'elles étaient pensées dans un monde avec une seule mémoire du point de vue de la sémantique et non deux. C'est le frein principal qui nous a poussés à ne pas mettre en place cette solution durant le stage.

4.1.3 L'extension WasmGC

Avant l'intégration de WasmGC au standard, un des inconvénients de Wasm était qu'il ne disposait pas, de manière native, de structures de données de haut niveau comme des tableaux. L'interfaçage d'un programme WebAssembly nécessitait souvent la copie de données dans sa mémoire linéaire, une opération potentiellement coûteuse. Les applications cryptographiques subissaient ainsi deux copies superflues : la première lors du transfert des octets du langage hôte vers la mémoire linéaire de Wasm, la seconde pour récupérer les résultats. WasmGC élimine ces copies en introduisant des structures et tableaux natifs en WebAssembly. Le langage hôte instancie ainsi des tableaux directement exploitables par Wasm. Les données sont alors copiées et traitées au sein de ces tableaux, éliminant tout transfert redondant. Concernant les algorithmes cryptographiques, il devient possible de stocker les octets dans un tableau WebAssembly dès la lecture de la ligne de commande, évitant toute copie superflue.

L'intégration de ces fonctionnalités dans Jasmin a nécessité l'introduction du type référence. Les références sur des tableaux Wasm étant sémantiquement proches des *reg ptr*, nous exploitons le mécanisme d'annotation de Jasmin pour donner plus d'informations au compilateur sur certaines parties du programme. Il suffit d'annoter les *reg ptr* avec une balise *ref* afin de guider le compilateur lors des différentes passes.

En pratique, c'est le cas uniquement lors de l'allocation de la pile. En Jasmin, un tableau est juste un pointeur sur la pile et les accès sont des lectures ou des écritures en mémoire. Dans le cas des références, nous voulons conserver la forme tableau puisque c'est comme cela qu'elles vont être traduites en WebAssembly. Il suffit d'ajouter à la structure de données interne représentant les *reg ptr* un champ booléen indiquant s'il s'agit ou non d'une référence. En initialisant ce champ grâce aux annotations attachées aux différents *reg ptr* du programme, dans cette passe de compilation, les vérifications mises en place sur les *reg ptr* (de sûreté par exemple) peuvent être conservées, puis la transformation en pile peut être annulée lorsqu'il s'agit en réalité d'une future référence Wasm.

Pour traduire ces tableaux en WebAssembly, un nouveau type doit être déclaré pour chaque type de tableaux existants et compatibles avec Wasm, c'est-à-dire des tableaux d'entiers 8, 16, 32, 64 ou 128 bits. Une fois ceci fait, il est naturel d'utiliser les instructions de lecture et d'écriture de WebAssembly pour traduire les lectures et écritures présentes dans le programme source. À l'instar des lectures et

écritures 8 ou 16 bits en mémoire, certains motifs introduits par les références doivent être reconnus. Par exemple, lire dans un tableau 8 ou 16 bits renvoie un entier 32 bits en Wasm, il faut donc transtyper le résultat de la lecture vers un 32 bits dans le programme source Jasmin afin de compiler ces lectures vers WebAssembly. C'est également le cas des écritures dans des tableaux 8 ou 16 bits qui attendent un entier 32 bits, il faut donc transtyper l'entier 32 bits en un entier 8 ou 16 bits afin de compiler ces écritures vers Wasm. Il ne faut pas oublier d'exporter des fonctions WebAssembly permettant à l'utilisateur de manipuler des tableaux depuis l'environnement externe. Par exemple, pour chaque type de tableaux, nous devons fournir une fonction permettant de créer un tableau, d'y lire ou d'y écrire et une dernière permettant de récupérer sa taille. La figure 6 illustre la compilation des références pour les motifs mentionnés.

```
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)))
  ;; Définition de $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 de get_u_array_i8, get_s_array_i8, set_array_i8, new_array_i8, ...
)
```

Fig. 6 : La compilation, de Jasmin à WebAssembly, de certains motifs de lecture et d'écriture dans des références

4.1.4 Méthodes de test

Le *backend* n'étant pas implémenté en Rocq et donc pas encore prouvé à ce stade, nous avons décidé de mettre en place une suite de tests afin d'avoir un peu plus de certitudes quant à la correction du *backend*. Pour ce faire, il y a deux grands types de tests.

Le premier consiste en des tests unitaires pour les instructions spécifiques à Wasm, donc pour les instructions SIMD en pratique. Ces instructions ne peuvent pas être compilées par les autres *backends*, c'est donc le seul moyen de les tester directement.

Le second consiste en des tests différentiels avec le *backend* x86-64. La correction de ce dernier est formellement vérifiée en Rocq. Il est ainsi possible de comparer, pour des entrées arbitraires, les résultats obtenus par les deux *backends* en sachant que le *backend* x86-64 renvoie forcément le bon résultat. Ces tests différentiels suivent deux axes : d'une part, la suite `common` de Jasmin valide les instructions génériques ; d'autre part, l'exécution d'algorithmes cryptographiques réels permet de valider indirectement les instructions SIMD en remplaçant les primitives x86-64 par leurs équivalents WebAssembly.

4.2 *Benchmark*

Cette section présente les différents *benchmarks* effectués sur le code Wasm généré par le nouveau *backend*. Nous avons d'abord comparé ses performances avec celles du *backend* x86-64, puis également testé l'efficacité d'interfacer le code Jasmin compilé vers Wasm avec d'autres langages.

4.2.1 Comparaison avec le *backend* x86-64

Le *backend* nous donne un fichier textuel au format `.wat`. Une fois le code WebAssembly obtenu, il est possible de l'interpréter. Toutefois, pour des questions de performances, l'approche usuelle consiste à l'exécuter dans un environnement externe. Pour ce faire, le fichier textuel `.wat` doit être compilé en un fichier binaire `.wasm`. Ce traitement peut être confié à un outil comme `wasm-as` qui va traduire littéralement les instructions en binaire ou bien `wasm-opt` qui dispose de quatre niveaux d'optimisation. Avec ce dernier, les programmes Wasm seront optimisés avant d'être traduits en binaire. Le fichier `.wasm` est ensuite connecté avec l'environnement externe, le plus connu étant JavaScript. Pour intégrer le programme WebAssembly dans JavaScript et l'exécuter, il faut faire appel soit à Node.js (un environnement d'exécution JavaScript), soit directement à un navigateur. Le code Wasm n'est plus interprété mais est finalement compilé à la volée vers du code x86-64.

Le code JavaScript et, par extension, le code WebAssembly utilisé profitent de la compilation JIT (*Just-In-Time*) pour rendre le code très efficace. Un JIT va d'abord compiler le code Wasm vers du code x86-64 naïf dont les performances sont très limitées, néanmoins dans le même temps, il lance une analyse plus complexe qui va donner du code x86-64 très efficace. Une fois cette analyse terminée, le code naïf sera remplacé par le code efficace. De plus, grâce à un système de *monitoring*, le JIT analyse l'exécution en temps réel, ce qui lui permet de raffiner le code optimisé obtenu pour le rendre encore plus efficace dans le contexte dans lequel il est utilisé.

Node.js et les navigateurs basés sur Chromium (Brave, DuckDuckGo, Google Chrome, Microsoft Edge, Opera, …) profitent du moteur V8 et Mozilla Firefox, quant à lui, utilise le moteur SpiderMonkey. Ce sont les deux principaux moteurs de JIT utilisés en pratique. Durant les *benchmarks*, nous essayerons de déterminer lequel est le plus efficace selon le contexte.

Afin de vérifier si l'efficacité du code WebAssembly produit rivalise avec celle du *backend* x86-64, nous avons évalué plusieurs algorithmes cryptographiques réalistes écrits en Jasmin. Les algorithmes utilisant des instructions AVX (SIMD en Wasm) sont optimisés et destinés uniquement à x86-64, et ne peuvent donc pas être compilés directement vers WebAssembly car ils utilisent des *intrinsics* x86-64. Les autres sont des versions de référence, c'est-à-dire qu'ils n'utilisent que très peu d'instructions x86-64 (hormis Gimli qui n'en utilise aucune).

Pour réussir à compiler ces algorithmes vers Wasm, nous avons simplement remplacé les instructions x86-64 par des instructions Jasmin ou WebAssembly : ce sont les versions minimales. Néanmoins, ces programmes étant initialement destinés à x86-64, certains choix d'implémentation ne sont pas du tout optimaux pour Wasm. Nous retrouvons notamment les adresses stockées sur 64 bits (qui devront être tronquées sur 32 bits en Wasm), des variables *stack* pour rendre l'allocation de registres faisable alors qu'il y a un nombre de variables locales illimité en WebAssembly ou encore des sauvegardes de variables sur la pile pour faciliter l'allocation de registres alors qu'il n'y en a pas pour Wasm. Ainsi, nous avons réécrit une autre version intégrant ces optimisations architecturales pour obtenir de meilleures performances en WebAssembly : ce sont les versions optimisées.

Pour mesurer les performances des versions x86-64 et Wasm nous répétons les algorithmes un nombre suffisamment grand de fois pour obtenir des temps de calculs compris entre 5 et 10 secondes pour chaque version. De plus, nous répétons l'expérience 20 fois afin d'atténuer la variance entre chaque mesure. Nous fixons également le cœur du processeur à utiliser pour stabiliser au maximum les mesures. Enfin, avant d'effectuer les vraies mesures, nous itérons 10% des répétitions dans le vide afin d'éviter les premiers temps peu stables et permettre au JIT d'obtenir la version optimisée du programme.

Les figures 7 et 8 présentent, respectivement pour les versions minimales et optimisées, les résultats du *benchmark* en comparant le ratio entre le meilleur temps d'exécution du *backend* Wasm (parmi les différents niveaux d'optimisation et de moteurs JIT) et le temps d'exécution du *backend* x86-64.

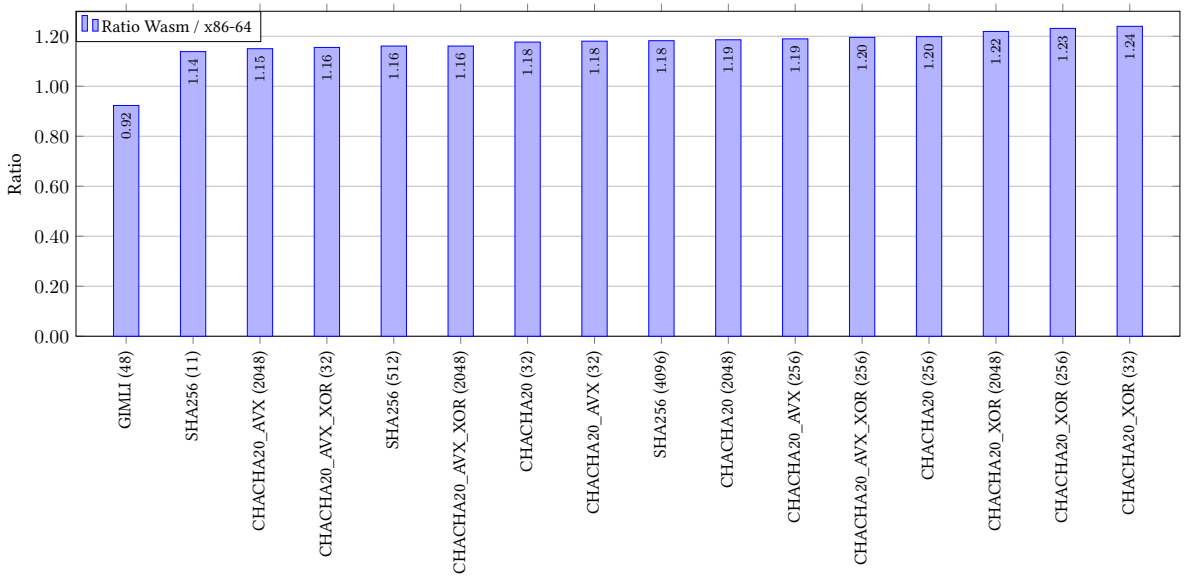

Fig. 7 : Rapport entre le temps d'exécution sur les versions minimales entre Wasm et x86-64

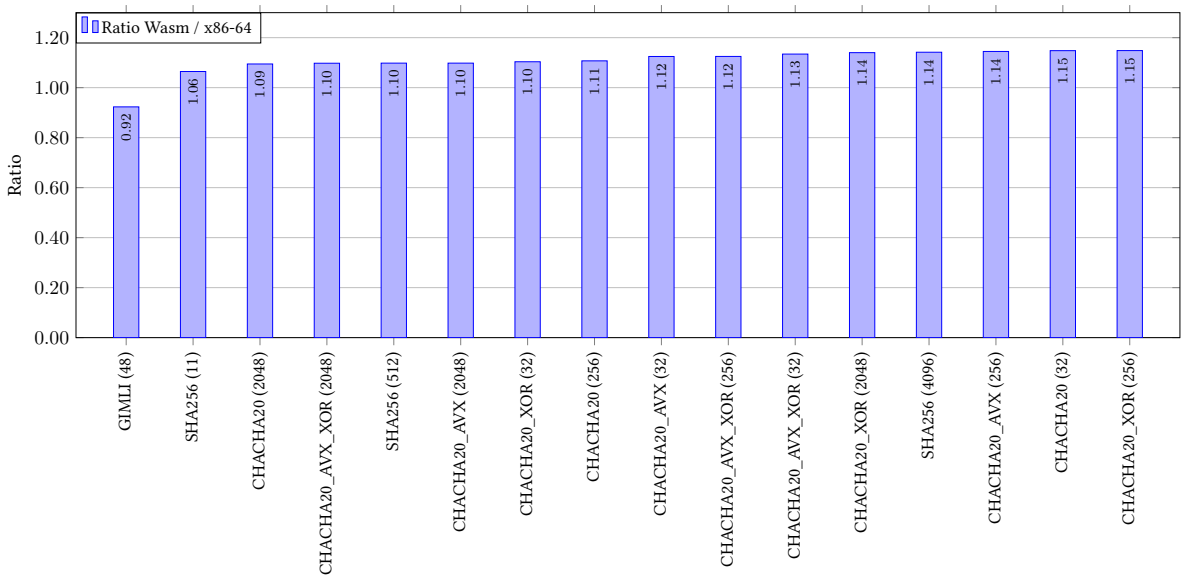

Fig. 8 : Rapport entre le temps d'exécution sur les versions optimisées entre Wasm et x86-64

Observons d'abord que le *backend* WebAssembly est légèrement plus efficace d'environ 8% pour Gimli, qui est le seul programme exactement identique pour les deux *backends*. Cette efficacité peut s'expliquer par le nombre inférieur d'optimisations reçues par x86-64. En effet, le *backend* Wasm reçoit les optimisations du compilateur Jasmin (comme pour x86-64) mais peut également recevoir les optimisations de `wasm-opt` ainsi que celles du JIT. Ainsi, sur un même programme (au niveau de Jasmin), le code généré par le *backend* WebAssembly peut être plus efficace à l'exécution. Ensuite, pour les autres algorithmes, le *backend* x86-64 est plus efficace avec environ 14% à 24% de surcoût pour les versions minimales contre environ 6% à 15% pour les versions optimisées. Les temps

obtenus pour les versions minimales sont tout à fait convenables pour des programmes pensés à la base pour x86-64. Ceci permet de prototyper avec Wasm rapidement à partir de programmes Jasmin pour x86-64 déjà existants avec des temps d'exécution réalistes. Néanmoins, les versions optimisées mettent en évidence une amélioration globale et logique des temps d'exécution. Ce gain découle directement de la spécialisation des programmes à l'architecture WebAssembly, tout en pouvant résulter de leur structure même. Il est possible qu'avec cette spécialisation, les programmes obtenus ressemblent davantage aux exemples utilisés lors du développement de `wasm-opt` ou des JIT, ce qui permet de mieux optimiser les programmes Wasm générés. Cette hypothèse reste néanmoins à nuancer, car l'écart de performance n'est pas suffisant pour exclure l'effet de la seule spécialisation du code, et dans le même temps, les programmes générés par Jasmin peuvent globalement ne pas être de la forme attendue.

Intéressons-nous maintenant à l'impact de V8 et SpiderMonkey sur les performances. Nous allons présenter uniquement les résultats sur les versions optimisées puisque les tendances sont relativement les mêmes qu'avec les versions minimales et que l'esprit de Jasmin est d'avoir une version de chaque algorithme spécialisée à l'architecture ciblée. Il est également possible que les mesures soient plus indicatives en utilisant des programmes WebAssembly plus réalistes. La figure 9 présente les résultats du *benchmark* en comparant le ratio entre le meilleur temps d'exécution obtenu avec V8 et le meilleur temps d'exécution obtenu avec SpiderMonkey.

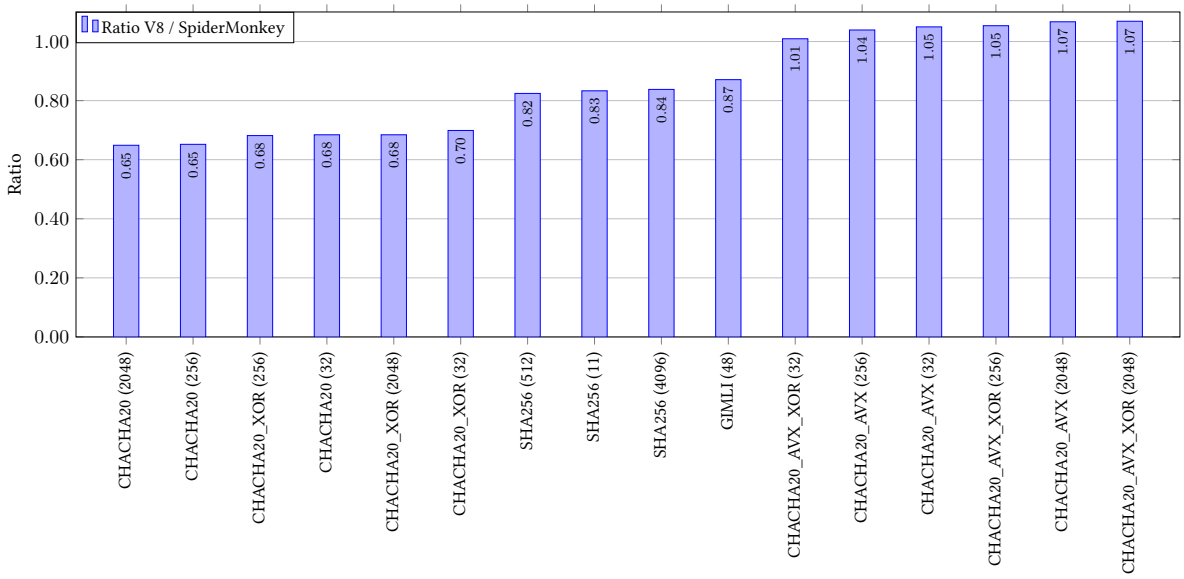

Fig. 9 : Rapport entre le temps d'exécution sur les versions optimisées entre V8 et SpiderMonkey

V8 surpasse clairement SpiderMonkey sur les algorithmes non vectorisés avec des ratios allant de 0,65 à 0,87, tandis que SpiderMonkey s'avère légèrement supérieur sur les instructions vectorisées avec des ratios allant de 1,01 à 1,07. Il y a alors potentiellement deux explications. La première serait que V8 est bien plus performant que SpiderMonkey mais que les techniques utilisées par SpiderMonkey (différentes de celles utilisées par V8) fonctionnent particulièrement bien avec les instructions vectorisées. L'autre serait alors de penser que le développement de V8 n'a pas été poussé sur les instructions vectorisées et ainsi que V8 aurait un retard sur ces dernières par rapport aux autres instructions. En effet, les instructions vectorisées n'étaient initialement pas présentes au début de Wasm, et sont devenues disponibles sous forme d'une extension (qui aujourd'hui fait partie intégrante du standard). Ce qui ressort de cette comparaison entre V8 et SpiderMonkey est que pour obtenir les performances les plus proches de celles de x86-64, il faut utiliser V8 lorsque le programme

n'est pas basé sur des instructions vectorisées et SpiderMonkey lorsque le programme est basé sur des instructions vectorisées. En ce sens, V8 et SpiderMonkey semblent être complémentaires.

Nous pouvons conclure de ce *benchmark* que le code WebAssembly obtenu est très efficace avec des performances similaires au *backend* x86-64 (seulement 6% à 15% de surcoût). Ce qui signifie que ce *backend* peut être utilisé en pratique afin d'exécuter des algorithmes cryptographiques efficaces et sûrs notamment sur le Web.

4.2.2 Interfacer Wasm avec d'autres langages

Après avoir testé, sur des algorithmes cryptographiques, les performances du code WebAssembly produit en les comparant à celles du code x86-64, nous avons décidé d'élargir le domaine d'application traité, notamment pour remplacer des bouts de programme écrits en JavaScript. Par exemple, le traitement d'image en temps réel dans le navigateur est très coûteux et JavaScript a du mal à suivre la cadence dans certains cas.

Nous nous sommes donc intéressés à des algorithmes de traitement d'image. Le premier applique un reflet flou à l'image, tandis que le second implémente un produit de convolution par matrice (noyau) pour accentuer les contours ou la netteté.

Notre approche consiste à prendre le code JavaScript et à le réécrire dans la syntaxe de Jasmin en suivant, le plus fidèlement possible, la logique du code initial sans appliquer la moindre optimisation. Ensuite, nous écrivons une deuxième version de cet algorithme en Jasmin qui essaye de tirer avantage des opérations vectorisées présentes en Jasmin et en Wasm.

Pour les différents résultats à venir, nous avons mesuré seulement le temps écoulé pour calculer le filtre et non la partie du programme récupérant les différentes données des images communes à JavaScript et WebAssembly. Pour l'algorithme effectuant un reflet, observons sur la figure 10 que la version de référence (i.e. sans vectorisation, sans optimisation) écrite en Jasmin est déjà 2,09 fois plus efficace que celle écrite en JavaScript pur. Cela peut paraître surprenant puisque deux copies sont effectuées pour transmettre les données de l'image à la partie Wasm et pour ensuite récupérer le résultat ; deux copies absentes dans la version JavaScript pur. Néanmoins, le temps gagné est suffisamment grand pour amortir le coût des copies grâce à la sémantique plus directe de WebAssembly qui permet notamment au JIT de mieux optimiser le code Wasm par rapport au code JavaScript. La vectorisation permet d'améliorer les performances pour atteindre un ratio de 2,80. Pour l'algorithme basé sur les noyaux, nous avons décidé de calculer pour chaque image les deux transformations. La figure 10 met en évidence que la version de référence est déjà 77 fois plus efficace que la version JavaScript pur, toujours en payant les deux copies supplémentaires. En ajoutant de la vectorisation, un ratio impressionnant de 212 est atteint.

Algorithme	Référence	Vectorisée
Reflet	2.09	2.80
Noyaux	77.04	212.04

Fig. 10 : Rapport entre le temps d'exécution de JavaScript et WebAssembly pour les traitements d'image

Ces résultats s'expliquent en partie par le fait que l'algorithme de réflexion effectue des calculs plus simples et de manière beaucoup moins intensive, notamment avec un temps de calcul moyen par image de 1,21ms pour JavaScript contre 0,43ms pour WebAssembly (version vectorisée). Par comparaison, le calcul des deux transformations utilisant un noyau est beaucoup plus coûteux

avec un temps de calcul moyen par image de 682,68ms pour JavaScript contre 3,22ms pour Wasm (version vectorisée). La sémantique de JavaScript impose beaucoup d'indirections et de tests dans les implémentations des interpréteurs ou des JIT contrairement à celle de WebAssembly qui est beaucoup plus simple et directe (notamment grâce à son langage fortement typé) offrant ainsi bien plus de perspectives d'optimisation. Cela confirme ce fait relativement connu dans la littérature, à savoir que Wasm est un candidat idéal pour remplacer les calculs intensifs effectués en JavaScript.

Expliquons pourquoi l'ajout d'*intrinsics* est important pour obtenir des performances optimales. La figure 11 présente l'implémentation en Jasmin de l'opération minimum vectorisé sans utiliser d'*intrinsic* ainsi que le code WebAssembly généré. La fonction `min_4s32`, pour chaque entier 32 bits de `a` et `b`, récupère le minimum et le stocke dans le vecteur résultat. Dans un programme Jasmin réaliste, la fonction `min` serait déroulée dans le corps de `min_4s32` qui serait elle-même déroulée dans le reste du code. Malgré le déroulement qui rend le code généré plus efficace, le grand nombre d'instructions utilisées pour calculer le minimum ralentit significativement le programme. En effet, c'est l'utilisation de l'*intrinsic* `#MIN_4s32`, générant l'instruction native Wasm correspondante, qui permet d'obtenir le ratio de 212 contre environ 90 sans.

```
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 pour i = 1, 2 et 3
    (return (local.get $res.335))
  )
  (export "min_4s32" (func $min_4s32))
)
```

Fig. 11 : Une implémentation en Jasmin du minimum vectorisé sans utiliser d'*intrinsic* ainsi que le code Wasm généré

5 Conclusion

5.1 Synthèse

Nous avons tout d'abord développé en OCaml un *backend* complet pour WebAssembly. En nous appuyant sur une architecture Wasm formalisée en Rocq, nous avons pu capitaliser sur le *frontend* existant. Ce *backend* intègre des passes de compilation spécifiques qui adaptent le programme source avant sa traduction finale. Toutes les instructions présentes dans Jasmin sont supportées ainsi que certaines instructions spécifiques à WebAssembly comme les instructions SIMD.

Nous avons également étendu le langage Jasmin afin d'introduire un nouveau type : les références. Ce nouveau type a permis d'utiliser une nouvelle extension WasmGC permettant notamment d'interfacer le code généré avec des langages à ramasse-miettes de plus haut niveau tels que Dart, Haskell, OCaml, Ruby ou Scheme.

Afin de consolider la confiance dans la correction du nouveau *backend*, une suite de tests a été mise en place. Elle est composée de tests différentiels pour les instructions Jasmin communes à toutes les

architectures, mais également de tests unitaires pour les instructions spécifiques à WebAssembly.

L'évaluation comparative des *backends* x86-64 et Wasm révèle des résultats encourageants : le surcoût mesuré n'est que de 10% à 15% sur des programmes cryptographiques Jasmin équivalents.

Nous avons, par ailleurs, écrit des programmes Jasmin spécialisés pour le *backend* WebAssembly pour réaliser du traitement d'images. Afin d'optimiser ces développements, nous avons intégré à Jasmin des instructions Wasm spécifiques supplémentaires. Les *benchmarks* confirment qu'en situation de calcul intensif, leur usage procure un gain d'efficacité significatif par rapport à JavaScript.

5.2 Limitations

La première limitation réside dans notre gestion de la mémoire dans les programmes WebAssembly obtenus. Idéalement, la mémoire source de Jasmin et la mémoire externe devraient être strictement séparées. Ce n'est pas le cas actuellement : la représentation intermédiaire du code ne permet pas de distinguer l'origine des accès. Lever cette limitation nécessiterait de modifier le compilateur pour assurer la traçabilité de ces accès.

Une seconde limitation, plus restrictive, concerne l'absence de preuve de correction fonctionnelle du *backend*. Son implémentation actuelle en OCaml fait obstacle à sa vérification formelle, rendant nécessaire sa réimplémentation en Rocq.

5.3 Travaux futurs

Un premier objectif serait de réimplémenter le *backend* directement en Rocq afin de prouver que la compilation vers WebAssembly préserve la correction fonctionnelle des programmes Jasmin. Ce travail s'appuiera sur une sémantique formelle de WebAssembly intégrant les instructions SIMD et l'extension WasmGC.

Une autre piste serait de prouver que toute la chaîne de compilation préserve la propriété cryptographique *constant-time*. Cela consisterait à viser l'extension *constant-time* de Wasm comme cible de compilation.

Il convient également de s'interroger sur l'impact de la compilation JIT sur la propriété *constant-time*. Ce mode de compilation est en effet susceptible de compromettre la résistance aux attaques par canaux cachés, à l'image des optimisations menées par les compilateurs traditionnels. Sa désactivation étant exclue pour des raisons d'efficacité, l'enjeu sera de concevoir une méthode contraignant le compilateur JIT à préserver la propriété *constant-time*.

5.4 Remerciements

Je tiens à remercier toute l'équipe du laboratoire pour son accueil chaleureux, son hospitalité et sa bienveillance. Je remercie tout particulièrement Jean-Christophe Léchenet pour son aide précieuse et son expertise sur Jasmin. Enfin, je suis profondément reconnaissant envers Benjamin Grégoire et Manuel Serrano pour leur enthousiasme face au travail accompli, leur excellent encadrement et leur soutien tout au long de ce stage de recherche enrichissant.

Références

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.