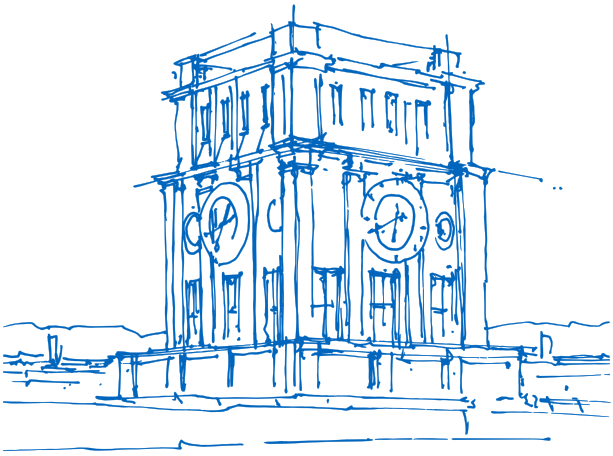
Chair for Security in Information Technology
TUM School of Computation, Information and Technology
Technical University of Munich

Cryptographic Implementations in Jasmin: High-Assurance Constant-Time CROSS

Konrad Winslow



Cryptographic Implementations in Jasmin: High-Assurance Constant-Time CROSS

Konrad Winslow

Chair for Security in Information Technology
TUM School of Computation, Information and Technology
Technical University of Munich

Cryptographic Implementations in Jasmin: High-Assurance Constant-Time CROSS

Konrad Winslow

Practical project

at the TUM School of Computation, Information and Technology of the Technical University of Munich.

Examiner:
Prof. Dr.-Ing.Georg Sigl

Supervisor:
M.Sc. Jonas Schupp

Submitted:
Munich, December 31, 2024

Abstract

Modern computer systems, like servers, microcontrollers, or edge devices, all require cryptographic schemes to secure their communication. Most schemes, as defined by standardizing organizations, are cryptographically secure: they cannot be compromised by analysis of public in- and outputs. However, a real-world application of a scheme requires more criteria, such as performance, side-channel resistance, and the soft- or hardware implementation to be error-free. Direct assembly-level programming is the most common way to fulfill these properties, because high-level languages such as C may introduce *side-channel leakage* of secret data and don't allow for fine-grained optimization to achieve maximum performance. While performance is measurable, programming errors of popular cryptographic implementations often go unnoticed for prolonged periods and compromise security once discovered. This can be both due to typical, logical errors in the code, but increasingly due to side-channel leakage that exposes secret values via some undefined interface of a device, for example from fluctuations in the power-consumption, the execution time, or memory accesses observable by an attacker. This matter is futher complicated by optimizing compilers which can introduce side-channel leakage that is not present in the source code of an algorithm.

Jasmin addresses this issue by allowing for the formal verification of cryptographic software. Jasmin comprises a toolchain for the development of verifiably correct programs: its programming language gives users fine control over the exact behavior that the compiled assembly will have, such as which registers are used to store intermediate values, and what exact machine-instruction is used for an expression. Jasmin's compiler is formally verified in Coq to produce machine-code that preserves the semantics of a program, meaning that the compiler won't introduce unintended leakage. Further, Jasmin programs can be extracted to EasyCrypt, a proof assistant for cryptographic schemes, that can be used to show the Jasmin program is cryptographic constant-time and functionally correct. This project implements the Codes and Restricted Objects Signature Scheme (CROSS) in Jasmin, verifies cryptographic constant time, and evaluates the toolchain. The main contributions of this work are a verified, high-assurance CROSS implementation usable as a library in other projects, a quantitative-, and a qualitative evaluation of Jasmin,

Contents

1 Introduction

With the widespread use of the internet, computer systems adopted cryptographic primitives to secure and attest communication. Because it is non-trivial to design secure cryptographic schemes, they are standardized by organizations and experts who assure a scheme meets a certain level of security and give recommendations on its usage, such as by fixing secure parameters. However, implementing said standardized algorithms is another potential source of errors: A bug or minor oversight can single-handedly compromise a system's security. Cryptographic algorithms are much more susceptible to implementation errors than conventional programs, because security can be compromised even if the implementation is functionally correct: Side-Channels can reveal secret information that would otherwise not be observable through defined interfaces.

Ensuring security of a cryptographic algorithm's implementation became even more difficult with the widespread adoption of IoT, edge computing, and other embedded devices. Attackers, easily able to physically access a device, can extract information via numerous side-channels that were inaccessible in a client-server setting. Side-channels often arise because of physical attributes of the machine that executes the algorithm. For example, certain microarchitectures of x86 can have longer memory access times based on the memory address, which leaks information, even in functionally correct programs. Because high-level programming languages obscure the underlying machine model, programmers still have to rely on assembly-level languages when writing side-channel resistant code. Assembly code is not portable between different instruction sets, and is much more costly to develop. Still, human error regularly leads to vulnerabilities in even the most used cryptographic libraries such as OpenSSL [38]. To make matters worse, the number of cryptographic schemes to implement for a system is increasing due to the modern requirement for post-quantum-secure cryptography [12].

Jasmin [20] offers an alternative to assembly for implementing cryptography that aims to mitigate aforementioned shortcomings. Jasmin's semantics are formalized in Coq [21], and it is formally verified that the compilation of Jasmin preserves these semantics, and does not introduce common types of side-channel leakage. Furthermore, Jasmin programs can be extracted as an EasyCrypt [18] model, which then allows reasoning about the Jasmin source code, such as proving constant-time properties, functional correctness, and safety of all memory accesses. Jasmin also offers a higher abstraction level than assembly, which makes code more portable between different instruction set architectures.

This project aims to evaluate Jasmin for the implementation of advanced cryptographic schemes and to lay out a structured approach for future usage. For this, the post-quantum secure CROSS [9] signature scheme is implemented in Jasmin. CROSS is a novel post-quantum secure, zero-knowledge signature algorithm, which is based on secure hashing and the syndrome decoding problem. Our implementation is verified cryptographic constant-time. Further, we use EasyCrypt to develop proofs for side-channel leakage- and functional correctness properties. Lastly, the Jasmin implementation of CROSS is benchmarked and compared with an existing CROSS implementation. The contributions of this project are: (i) A high assurance Jas-

min implementation of the CROSS signature scheme called CROSS-Jasmin, which can be used as a library in other systems; (ii) cryptographic constant-time verification of CROSS-Jasmin, which validates that the implementation does not leak secret data; (iii) a qualitative evaluation of the Jasmin toolchain in regard to usability, advantages and technical limitations, and (iv), a quantitative evaluation in the form of benchmarks.

1.1 Structure of this Work

The EasyCrypt framework for the modeling and formal, computer assisted verification of cryptographic algorithms is explained in chapter 2. The Jasmin language, features, toolchain, and its interaction with EasyCrypt are outlined in chapter 3. Chapter 4 describes the design and realization of this project's high assurance CROSS implementation 'CROSS-Jasmin'. It introduces the reader to working with Jasmin in a practical setting, what pitfalls are likely to arise, and how it differs from typical high-level language workflows. The results of the project are discussed in chapter 5, including a discussion on advantages and limitations of Jasmin and the implementation's performance in comparison to the CROSS-C implementation. It also references alternative languages and tools to Jasmin. Chapter 6 concludes the project and gives a perspective on future work.

2 EasyCrypt

EasyCrypt [10, 18] is a framework for the modeling of cryptographic algorithms, development of security proofs, and their automated, computer assisted validation. It was initially designed to allow for game-based security proofs, which formalize adversaries playing "games", and shows security by bounding the probability of events below certain thresholds [10, 36]. While there were pre-existing tools for computer-checked game-based proofs at the time, EasyCrypt aimed to allow for easier proofs, requiring less effort than in alternative frameworks. Algorithms and games are modeled as imperative, stateful procedures in EasyCrypt, which have a shared global memory and local variables. The procedures may query random oracles, whose statistical distribution can be user-defined. Security can then be shown by relating the probabilities of two procedure's states to each other using some relation: for example showing that distinguishing two plaintexts is just as likely as factoring numbers [10]. In addition to proof methods supported by EasyCrypt itself, it integrates with off-the-shelf SMT solvers, such as Z3 [27] or CVC5 [15], to discharge certain goals.

2.1 EasyCrypt Specifications

EasyCrypt's specification language allows users to declare and define, variables, total functions, data types, axioms/lemmas, and imperative, stateful modules. Total functions, variables, and data types are treated as normal expressions within EasyCrypt, while modules model cryptographic algorithms and become the subject of proofs and lemmas. As such, they cannot be evaluated by EasyCrypt and are instead manipulated within proofs. The following sections will give a brief overview of the specification language.

2.1.1 Expression Language

Listing 2.1.1 shows the definition of the algebraic datatype *list*, parameterized by the type *'a*. Functions on custom and built-int data types can be declared with the *op* keyword; these may be recursive, but termination must be provable by EasyCrypt. Expressions also support lambda-functions, and built-in operators can be user-defined on custom data types. Any declared variables, types, and functions, can be used in procedural specifications and lemmas.

Types, variables, and operations may be left abstract. In that case, one only declares a name and type, but omits a definition. Axioms allow reasoning about instances of abstract members. For example, listing 2.1.1 declares an abstract monoid type, operations on it, and defines it via axioms. Because axioms are assumed to be correct by EasyCrypt, care must be taken not to introduce false or inconsistent statements.

Definitions, lemmas, proofs, and modules can be packaged as EasyCrypt theories, which allows for their composition and reuse. Listing 2.1.2, for example, shows the definition of the abstact theory *MonoidDI*, which is later instantiated with the type *realp*. Any axioms assumed

```
type 'a list = [Nil
  | Cons of 'a & 'a list].
op len ['a] (acc : int, xs : 'a list)
  : int =
  with xs = Nil => acc
  with xs = Cons y ys
    => len (acc + 1) ys.
op length ['a] (xs : 'a list)
  : int = len 0 xs.
op xs = Cons 0 (Cons 1 (Cons 2 Nil)).
op n : int = length xs.
```

```
type monoid.
op id : monoid.
op (+) : monoid -> monoid -> monoid.
axiom LeftIdentity (x : monoid) :
  id + x = x.
axiom RightIdentity (x : monoid) :
  x + id = x.
axiom Associative (x y z : monoid) :
  x + (y + z) = (x + y) + z.
```

Listing 2.1.1 Functional EasyCrypt specification example.

by the abstract theory must be proven at instantiation. The standard library of EasyCrypt includes multiple abstract and concrete theories that can be imported by users.

```
abstract theory MonoidDI.
  clone include MonoidD.
  axiom addmI: right_injective (+).
  lemma mul0m_simpl x : zero * x = zero by apply mul0m.
  lemma mulm0_simpl x : x * zero = zero by apply mulm0.
end MonoidDI.
clone include MonoidDI with
   type t  <- realp,
   op zero <- of_real 0.0,
   op MulMonoid.one  <- of_real 1.0,
   op ( + ) <- Rp.( + ),
   op ( * ) <- Rp.( * )
proof * by smt(of_realK to_realP to_real_inj).
```

Listing 2.1.2 EasyCrypt theory definition and instantiation [18, Xreal.ec].

2.1.2 Module System

Modules can be abstract or concrete, and contain procedures as well as global variables. Each module has a module type that can be declared separately. Module types allow for composition: A module can be parameterized by a module type, and later instantiated with a module having the correct type. Procedures consist of local variables and a sequence of statements, which may be procedure calls, variable assignments, random assignments, while loops, or if/else branches. Statements in a procedure can have side effects by modifying global variables. Within procedures, members of other abstract and concrete modules can be referenced. Listing 2.1.3 is an example of modules in EasyCrypt: The module Z could be instantiated with either *Y1* or *Y2*. Semantically, the global state of modules are merged into one when they interact with each other through shared variables or procedure calls. If Z is not instantiated with a defined module, X would be left abstract. This is useful for adversarial security proofs, where the

adversary is left as an abstract module. When used in lemmas, X is universally quantified, which means that *f()* could perform any action, such as modifying Z's global variable *y*. An example of using abstract modules to model adversaries that attempt to distinguish ciphertexts is given by [10], who model a game between two adversaries as a concrete module representing the game, parameterized by two abstract modules representing the adversaries.

```
module type X = {
  proc f() : unit
}.
module Z(X : X) = {
  var y : int
  proc f() : unit = { X.f(); }
}.
```

```
module Y1 = {
  var y, z : int
  proc f() : unit = { y <- 0; }
  proc g() : unit = { }
}.
module Y2 = {
  var y : int
  proc f() : unit = { Y1.f(); }
}.
```

Listing 2.1.3 EasyCrypt module system example [19].

2.1.3 Probability distributions

EasyCrypt supports discrete probability distributions as data types to express security assumptions and guarantees in lemmas, and to model random oracles that can be queried during cryptographic games. The user can define distributions on a discrete set using the *dist* keyword. Distributions need to be defined by their mass-function. Using the built-in operator *mu* and predicates like *is_lossless* or *is_uniform*, users can axiomatize the mass-function. For example, *is_lossless* is defined as `pred is_lossless ['a] (d : 'a distr) = mu d (fun (x : 'a) true) = 1`, which says that the sum of probabilities for any element *'a* is 1.

Procedures can use random oracles by sampling from a probability distribution using the *<$* operator. Logically, sampling transforms the current distribution of the global memory according to the sampled distributions' mass function.

2.2 EasyCrypt Proofs

Lemmas in EasyCrypt are proven by a sequence of *tactics*. Tactics embody general reasoning principles and transform the current proof state's goal into new subgoals that, according to the tactic, are sufficient for the original goal to hold. Each goal consists of a context that has two parts: a conclusion that we are trying to prove, and a set of variables and assumptions that we may assume true to prove the conclusion. To those familiar, goals in EasyCrypt can be read similarly to a logical rule of inference that we are trying to prove. Because EasyCrypt proofs are in "tactic-style" [30, p. 259], they are harder to understand and lack structure to human readers compared to conventional proofs.

2.2.1 Ambient logic

EasyCrypt includes an ambient higher-order logic for proving general mathematical judgments, and does not reason about the results of imperative procedures, which are handled by

logics described in sec. 2.2.2. In addition to aforementioned tactics, ambient logic goals can be automatically discharged by the help of SMT solvers, for which EasyCrypt uses Why3 [37].

Formulas of the ambient logic extend expressions by adding quantifiers, predicates, memories, and judgments about imperative programs. Formulas are used to state axioms and lemmas, with axioms assumed to be true, while lemmas need to be proven via tactics. Users have to ensure axioms are consistent, because otherwise they can allow one to prove false lemmas: For example the axiom `axiom Empty: !(exists (x : t), true)` can be used to prove *false*, since all types in EasyCrypt are assumed to be non-empty.

The following is an overview of some ambient logic tactics [19]:

- τ `=>` $\iota_1 ... \iota_n$ (*Introduction*): Moves assumptions from the goal's conclusion to the goal's assumption. τ can be another tactic, in which case the introduction pattern ι_i is matched with subgoal G_i after applying τ.
- τ `:` $\pi_1 ... \pi_n$ (*Generalization*): Moves assumptions $\pi_n .. \pi_1$ from the goal's assumption to the goal's conclusion, adding implications or universal quantification, and then run tactic τ.
- `exists` e: Transforms a goal's conclusion of the form `exists (x : 'a), P x` into `P e`.
- `smt`: Tries to solve the goal using SMT solvers.
- `apply` p (*Backwards reasoning*): Tries to unify the goal's conclusion with the conclusion of p, and replace it with p's assumptions.
- `case` ϕ (*Case analysis*): Perform an excluded-middle case analysis on ϕ. If ϕ is not given, destruct the goal's conclusion depending on form. For example, inductive datatypes are destructed into their constructors, while a disjunction is destructed into its subformulas.
- `elim /`L: Perform induction on inductive datatypes, the integers, or the induction principle L.

To illustrate, listing 2.2.1 contains an example. Using EasyCrypt's expression language, we defined an inductive datatype `list` and the `snoc` operation on it, appending an element to the back. Lemma `snoc_len` asserts that `snoc` makes any list one element longer. The blue-highlighted text has already been processed by EasyCrypt, with the current goal's context being displayed to the right. We start the proof by using the `elim` tactic to perform induction on l, subsequently moving `x0, l, IH` to the context's assumptions. The remaining conclusion is proven by applying the induction hypothesis `IH` and simplifying the goal.

2.2.2 Program logic

Ambient logic formulas can state assertions about imperative procedures in the following ways [19]:

- `Pr[M.p(`$e_1 ... e_n$`) @ $m : `ϕ`]` (*Probability expression*): Represents the probability that running procedure `M.p(`$e_1 ... e_n$`)` with global memory `$m` results in a global memory satisfying ϕ.

```
type ('a) list = [ | Nil
  | Cons of 'a & 'a list ].

op snoc ['a] (x: 'a, l: 'a list) =
  with l = Nil => Cons x Nil
  with l = Cons h t
    => Cons h (snoc x t).

lemma snoc_len ['a]
  (x : 'a, l: 'a list):
  length (snoc x l) = 1 + length l.
proof.
  elim l => // x0 l IH.
    + simplify. rewrite IH. simplify.
      trivial.
qed.
```

```
Current goal
Type variables: 'a

x: 'a
x0: 'a
l: 'a list
IH: length (snoc x l) = 1 + length l
------------------------------------
length (snoc x (Cons x0 l))
  = 1 + length (Cons x0 l)
```

Listing 2.2.1 Example ambient logic proof of snoc_len.

- `hoare[M.p : ϕ ==> ψ]` (*Hoare logic (HL) judgments*): Asserts that running `M.p` with precondition ϕ on parameters and the global memory, results in a memory satisfying the postcondition ψ.
- `phoare[M.p : ϕ ==> ψ] [> | < | =] e` (*probabilistic Hoare logic (PHL) judgments*): Like a *HL judgment*, but relating the probability of satisfying the postcondition after termination to e.
- `equiv[M.p ~ N.q : ϕ ==> ψ]` (*probabilistic relational Hoare logic (PRHL) judgments*): Relates the probability distributions of `M.p`'s and `N.q`'s global memories, after execution, to each other according to preconditon ϕ and postcondition ψ. Within the judgment, `$1` refers to the memory of `M.p`, and `$2` refers to the memory of `N.q`. The judgment is true iff. all memory pairs satisfying ϕ are transformed into a pair of distributions on memories Π_p, Π_q by running `M.p` and `N.q` on them, such that they satisfy ψ. The distributions satisfy ψ iff. there is a function f dividing the mass μ of each $m \in \Pi_p$ between the $m \in \Pi_q$ such that $\forall m_q \in \Pi_q.\mu(m_q) = \Sigma\{f(m_p)|\psi(m_q, m_p)\}$. PRHL judgments are especially relevant when modeling adversarial games.

Some background knowledge in Hoare logic is useful when trying to understand how EasyCrypt reasons about imperative programs, for example by reading [33] or [31]. Simplified, Hoare logic defines *Hoare triples*: $\{P\}c\{Q\}$, which means that program c, if executed in a state satisfying P, terminates in a state satisfying Q. What makes Hoare logic usable are proof rules that follow a programs structure and allow for the composition of proofs and Hoare triples [33]. There are rules for each type of statement encountered in a program, such as assignment, if/else statements or while loops, plus a *sequencing* and *consequence* rule. The sequencing rule states that a Hoare triple on a sequence of two statements $\{P\}c1; c2\{Q\}$ can be proven from $\{P\}c1\{R\}$ and $\{R\}c2\{Q\}$ for any Q. The rule of consequence meanwhile asserts that $\{P\}c\{Q\}$ can be proven from $\{P'\}c\{Q'\}$ and $P \Rightarrow P'$ and $Q' \Rightarrow Q$, informally weakening the precondition and strengthening the postcondition. A nice feature of Hoare logic is that it is easy to automate rule

application, except for while loops: A human has to state correct and sufficient loop invariants for each while-loop and prove inferences between conditions, and an algorithm can carry out the rest of a program proof [31][208-212].

EasyCrypt extends standard Hoare Logic by the notion of probability for PHL and PRHL judgments, which arises because of the sampling (`<$`) operator in programs. When proving statements about imperative programs, the goal's conclusion will be in the form of a HL, PHL, or PRHL judgment instead of an ambient logic formula. There are a number of specialized tactics to transform program judgments in the current goal and thus to allow reasoning about programs, such as:

- `proc`: replaces a judgment of a concrete procedure with its body.
- `wp`: reasons about the current program using the *weakest precondition* [31][205], consuming as many statements from the end of a procedure as possible.
- `rnd`: Consumes a randomly sampled assignment from the end of programs, replacing the conclusions postcondition with the probabilistic weakest precondition.
- `transitivity`: Proves a PRHL judgment `equiv[M.p ~ N.q : P ==> Q]` by the transitivity of new subgoals `equiv[M.p ~ T.r : P ==> R], equiv[T.r ~ N.q: R ==> Q]`.
- `byequiv`: Allows to reason about the probability of events based on the equivalence of procedures, or games. For example, using `byequiv (_ : P ==> Q)` proves a conclusion `Pr[M.p(a) @ &`m_1 `: `E_1`] = Pr[N.q(b) @ &`m_2 `: `E_2`]` with the new subgoals: `equiv[M.p ~ N.q : P ==> Q]`, a goal stating that precondition P holds on memories $\&m_1$ and $\&m_2$, and one stating that $Q \Rightarrow E_1\{\&m1\} \leftrightarrow E_2\{\&m2\}$

[19] contains a more complete list of program- and ambient logic tactics.

Using the program logics, "game-hopping" style security proofs can be constructed: PRHL judgments can be used to show equivalencies between a cryptographic algorithm and abstract security assumptions, while bounding probabilities state how likely it is for an attacker to break security. For instance, [10] prove security of Hashed ElGamal by first relating ciphertext distinguishability to the "Computational Diffie-Hellman assumption" using PRHL judgments, and then show that the probability of successfully distinguishing ciphertexts is bounded by the probability of breaking said assumption. Note that security proofs are not restricted to such "game-hopping" styles: [16], for instance, prove the zero-knowledge property modeling a simulator that is able to rewind a malicious verifier.

3 Jasmin

Jasmin [20] is a domain-specific programming language for implementing high-speed, high-assurance cryptographic software. Jasmin's toolchain consists of a verified compiler, a safety-checker, and a security type-checker. Jasmin's semantics are formalized in Coq [4], and the compiler is verified to produce assembly that preserves the source code's behavior and constant-time properties, although the proof that the compiler preserves timing attack mitigations has not yet been formalized in Coq [1]. Further, Jasmin programs can be extracted to EasyCrypt, which allows formal reasoning about the behavior and security properties of Jasmin programs.

Jasmin is designed to bridge the gap between high-level and "assembly-level" cryptographic development, and to produce programs with the following desired properties: (i) Efficiency, (ii) Side-channel resistance, and (iii) Functional correctness [4]. Because high-level languages typically don't achieve desired performance and can't guarantee side-channel resistance of the compiled code, cryptographic schemes are predominantly implemented in direct assembly [32]. Assembly-level code is efficient and can be side-channel resistant, but is unportable, costly to develop, and difficult to implement correctly. Further, verifying correctness, constant-time execution, and security properties is not easily applicable to assembly [4].

Jasmin addresses the shortcomings of traditional direct assembly development by mixing high-level and low-level language concepts, while also providing infrastructure for the formal verification of programs, whose assumptions carry down to the assembly generated by the compiler.

3.1 Language

The Jasmin language is a high-level imperative programming language that offers precise control over the generated assembly code inspired by qhasm [11]. Jasmin programs consist of high-level abstractions, such as parametric functions, for- and while-loops, implicit stack layout and numerical data types. Unlike typical high-level languages, Jasmin gives programmers the ability to refine statements and operators with exact machine instructions, and to control the register- and stack allocation of variables and flags. Being able to write code with similar advantages to direct assembly comes with some complexities: Jasmin programs have to explicitly handle register allocation and spilling, and all expressions have to be side-effect free. It is also not possible to reference or de-reference memory, although this stems from a limitation of the language semantics. All Jasmin code corresponds closely to the assembly generated from it, which the developers call *predictability* [4] of the language.

3.1.1 Top-level program

Jasmin programs consist of static parameters, global variables, and functions. Functions may be declared as `fn`, `inline fn`, or `export fn`. Functions can take an arbitrary number of argu-

ments and return multiple results, but arguments cannot be passed on the stack, requiring the allocation of fresh registers. Exported functions become global symbols in the final assembly code that can be linked against according to the platform's calling conventions. Next to global variables, which must be immutable, Jasmin supports lexically scoped, local variables within the body of functions. All variables are explicitly associated with a type and storage class. Static parameters on the other hand do not require a storage class and are available as compile-time constants. Each function's body is a sequence of statements and variable declarations, ending in a return statement.

Top-level program elements can be associated to a namespace. Namespaces offer the ability to separate components of a Jasmin program or library and a mechanism to reuse code with different parameters (e.g. listing 4.2.1). All elements in a namespace must be prefixed by the namespace's identifier when used outside it. Jasmin offers a simple textual inclusion directive, similar to C `#include` macros, which can be combined with a namespace to parameterize and reuse code.

3.1.2 Types

Jasmin supports the following types:

- `u8, u16, u32, u64, ...` (*Numerical types*): Only unsigned types are available. Numerical types must explicitly be cast between each other. Signed integer semantics are implemented as special operators on unsigned numbers.
- `bool` (*Boolean types*): Must be the result of an instruction that sets a flag as a side effect or a compile-time known value.
- $u_i[N] \mid i \in \{8, 16, 32, 64, ...\}$ (*Functional Arrays*): Arrays are a sequence of numerical values. The size of an array must be a static value, such as a static parameter. Jasmin's arrays behave as first-class values: when the array is passed as an argument, or assigned to another array variable, all the contents of the array are "moved" to that new location. Data is not actually copied in final assembly code because of optimizations, but at the source-code level values of an array cannot be accessed anymore after assigning it to a new location. This also requires all functions to return the arrays they modify.
- `int`: Compile-time integer values of arbitrary size.

Functional arrays are atypical for imperative, high-level languages and assembly-level programming, but simplify verification of Jasmin programs [1].

3.1.3 Storage classes

Storage classes must be explicitly specified for every variable and function parameter. This allows the programmer to manually handle register allocation and spilling when fine-tuning performance, and prevents the compiler from introducing side-channel leakage from implicit register-allocation or aliasing. Because the code declares which values are kept in machine-registers, it has to be written with a specific machine's register set in mind, as too many register-stored values cause a compilation error due to unallocated variables. The available

storage classes are: `reg` for register-allocated values, `stack` for stack-allocated values, `reg ptr` for register pointers to arrays, `stack ptr` for array pointers stored on the stack, and `inline` for compile-time constants. Note that the `reg/stack ptr` storage classes are not comparable to C-style pointers, as they still require to be attached to an array type with statically-known size.

3.1.4 Expressions

Expressions are used within statements, for example as the condition of a while-loop or the value of a variable assignment. Expressions are free of side-effects. They encompass: Binary- and unary operators, such as arithmetic-, logical, and comparison operators; variables; memory loads and array accesses; function calls; constants, and *intrinsic operators*. Intrinsic operators are special in that they can be architecture-dependent, and allow the use of exact assembly instructions within Jasmin programs. The use of intrinsic operators aims to make Jasmin a viable alternative to handwritten assembly code. Of course, using architecture-dependent intrinsics reduces the code's portability and makes it less maintainable. Operators and intrinsics are parameterizable by size- and sign-suffixes to specify the exact type of data they operate on. For example, listing 3.1.1 shows a Jasmin implementation that has been optimized for performance using machine-specific intrinsics. The reference version on the left operates on one 16-bit value, using plain binary addition. The optimized version on the right adds 16 values, each 16-bit, in parallel using the `PADD` assembly instruction.

With few exceptions, a Jasmin expression gets mapped to a single assembly instruction by the compiler. This is to achieve predictability and prevent unwanted side channels. For instance, an addition of two `u64` numbers gets mapped to an x86 `ADD` instruction with 64-bit register operands. However, if no instruction corresponds to the expression, like when trying to add three or more register operands, the compilation fails with an error.

3.1.5 Statements

Statements are the only construct that can alter the state of a Jasmin program via assignments or control flow. Assignments are of the form $d_1 \dots d_n = e$ for local variable assignments, $x[e_i] = e_j$ for array assignments, and $[x + e_i] = e_j$ for memory assignments. Array and memory assignments differ in that memory assignments operate on a machine-word sized address, while array assignments operate on fixed-size functional arrays. There is currently no way to convert between the two: memory addresses must be supplied to Jasmin programs from the environment, for example C code. While-, For-, and conditional statements allow the programmer to alter the control flow of the program. While-loops can only be used on run-time computable expressions, and for-loops require compile-time values, like `inline` variables or parameters, as bounds. Another difference between the two is that for-loops get unrolled by the compiler, duplicating the body. Conditional if/else statements with compile-time guards are also unfolded by the compiler, which allows users to approximate a macro system. Semantically, a memory assignment is the only statement that has an implicit side effect, since arrays are always passed and returned by-value. Listing 3.1.1 shows how the programmer explicitly shapes the compiled code when writing Jasmin: On the left, the while-loop gets compiled into a backwards branch, and the condition into an immediate comparison that operates on a 64-bit register storing `i`. The optimized code on the right instead becomes a linear sequence of the

loop's body, with no comparison or branch taking place at run-time, and no register being allocated to `i`.

```
fn add(reg ptr u16[KYBER_N] rp bp)
        -> stack u16[KYBER_N] {
  ...  reg u64 i;
  i = 0;
  while (i < KYBER_N) {
    a = rp[(int)i];
    b = bp[(int)i];
    r = a + b;
    rp[(int)i] = r;
    i += 1;
  }
  return rp;
}
```

```
fn add(reg ptr u16[KYBER_N] rp bp)
        -> stack u16[KYBER_N] {
  ... inline int i;
  for i = 0 to 16 {
    a = rp.[u256 32*i];
    b = bp.[u256 32*i];
    r = #VPADD_16u16(a, b);
    rp.[u256 32*i] = r;
  }
  return rp;
}
```

Listing 3.1.1 Reference [17, ref/poly.jinc] and vectorized [17, avx2/poly.jinc] Kyber [14] Jasmin functions.

3.2 EasyCrypt extraction

Jasmin programs can be extracted to EasyCrypt for verification of functional correctness, constant-time properties, and security guarantees [1]. The overall structure of a Jasmin program is naturally translated to EasyCrypt, as EasyCrypt already formalizes imperative modules, procedures, while-loops, conditional statements, and variables. In addition, Jasmin has an EasyCrypt library with theories that model the semantics of Jasmin's operators and memory. Memory is modeled as a global module variable containing a functional map between addresses and the bytes stored at them. When a Jasmin procedure contains a memory assignment, it is represented as updating the map with a new value in EasyCrypt. Because the entire Jasmin program shares a global memory, but variables in EasyCrypt exist per-module, the extracted functions are wrapped in a single EasyCrypt module. These extractions can be used to reason about functional correctness and security by using the same approaches discussed in ch. 2.

For verifying cryptographic constant-time, Jasmin programs extracted to EasyCrypt are instrumented with *leakage traces*. The EasyCrypt module representing a Jasmin program also includes a list of *leakages* as its own global variable. All branching conditions, accessed array indices, and accessed memory addresses are explicitly appended to the list of leakages, thus representing a "trace" throughout the program's execution. Cryptographic constant time can then be stated as follows: Given that two executions of an exported Jasmin function agree on public inputs, their leakage traces after termination are equal. This statement can be proven using existing EasyCrypt tactics and Hoare logic.

An example EasyCrypt extraction for an arithmetic primitive of CROSS is shown by listing A.3. Listing 3.2.1 is a formal EasyCrypt proof that the execution time of the "shake256" Jasmin function only depends on public inputs: It expresses a PRHL equivalence between two executions that start with equal leakages, input lengths, and addresses for the digest and input arrays, and shows that they must always terminate with the same leakage trace, thus

guaranteeing cryptographic constant time, because the secret content of the input array can differ.

```
equiv shake_ct :
  Export_ct.M.shake256 ~ Export_ct.M.shake256 :
    ={leakages, len, digest_out, arr} ==> ={leakages}.
proof.
  proc; inline *; sim.
qed.
```

Listing 3.2.1 Cryptographic constant-time proof of the "shake256" Jasmin function.

Existing work has also used Jasmin's EasyCrypt extraction to prove security guarantees on top of functional correctness or side-channel resistance. In particular, using PRHL, the equivalence between Jasmin code and formal EasyCrypt specifications can be proven. This shows that security guarantees of the specification also apply to the particular implementation in Jasmin, and its compiler guarantees those properties are maintained by the final assembly code. [4] uses such an approach to provide a verified implementation of SHA-3 [29].

3.3 Constant-time checker

Jasmin was extended with a security type system and a static analysis tool for checking cryptographic constant-time at a later release [35]. The security type system can also model misspeculation to protect against Spectre-V1. The constant-time checker is an alternative to manually showing side-channel resistance via Jasmin's EasyCrypt extraction.

The differences between both methods are shown in table 3.1. Other than being able to reason about speculation, the main advantage of the static analysis tool is the lower effort imposed on the user. Variables, parameters, and results can be annotated with a `#secret` or `#public`. The type system ensures that addresses and branches never depend on secret values. As any static analysis tool, the type-checker has to over-approximate the set of unsafe programs and may report false-positives. Using EasyCrypt to verify cryptographic constant-time of a Jasmin program needs more human involvement, but is also more flexible, in the sense that it is possible for users to state their own theorems and develop proofs that apply even when the type-checker could not verify the program.

The two approaches also differ when having *intended leakage* [5, 35]. Intended leakage occurs, for example, due to secret data-dependence of a hash- or otherwise irreversible function. These leaks are not security critical, since the leaked values can be considered public. However, to both the type system and the EasyCrypt leakage trace these values were computed from secret inputs, and are thus also considered secret. Probabilistic sampling, such as in Kyber or CROSS, is another form of intended leakage. The type system offers a `#declassify` annotation to "turn" secret values public, while EasyCrypt requires axioms and additional proof steps to deal with intended leakage.

Table 3.1 Comparison of Jasmin's constant-time type-checker and EasyCrypt verification.

	Type-checker	EasyCrypt verification
Speculative constant-time support	Yes	No
Probabilistic program method	Declassification	Manual proof [5]
One-way function method	Declassification	Axiomatization, manual proof
Flexibility	Low	High
Usability	High	Low

3.4 Safety-checker

Lastly, the Jasmin toolchain has a safety-checker using static analysis to verify a Jasmin program is memory-safe and terminates. Because Jasmin programs depend on external parameters, specifically memory pointers, the checker can't decide a program's safety based solely on the source code. Instead, the safety-checker outputs a *safety precondition*, which is a sufficient condition on input parameters guaranteeing safety. As with the static constant-time analysis, the safety-checker can output false-positives. Unlike the constant-time type-checker, the safety checker does not always terminate within a practical time because the analysis is more complex.

4 Implementing CROSS with Jasmin

Before this project, CROSS had a technical specification [9] and a proof-of-concept reference and optimized C implementation (CROSS-C) [8]. The technical specification details the mathematical background of the CROSS signature scheme, explains the parameter choices and versions, and includes a high-level procedural description of the key-generation, signature-generation, and signature-validation algorithms. It also explains why, in general, the signature scheme is constant-time and should not leak secret inputs. The proof-of-concept implementation is not validated for cryptographic constant-time, and because it is written in C, widely-used optimizing compilers can introduce additional leakages that are not present at the source-level. Both the optimized and reference code support all variants of CROSS described in [9], and are used for benchmarking and generation of Known Answer Tests [28].

To gain experience with the Jasmin toolchain and evaluate it for future use it suffices to focus efforts on implementing one variant of CROSS. The variant to implement and verify in Jasmin was chosen according to the following criteria: (i) It should cover all language features and potential pitfalls of Jasmin, (ii) it should be suitable for constant-time and functional verification, (iii) it should represent a realistic cryptographic scheme, and (iv) the final implementation can be compared to a reference via benchmarks. All of CROSS' variants fulfill criteria (ii) and (iii), and would only have marginal differences in criteria (i), since the algorithmic changes between them would not be interesting to implement on-top of what is already present in the shared procedures. However, it is important to have a complete, functional end-to-end implementation of CROSS to compare to CROSS-C using benchmarks, so the simpler variants are favored according to (iv). Based on the criteria, we chose CROSS-R-SDP fast: it requires less code and a functionally complete implementation is more likely to fit within this project's scope. Extending CROSS-R-SDP fast to the other variants is straightforward and amounts to changing datatypes and sub-functions used in the scheme. As described in ch. 3, Jasmin supports optimization by replacing expressions with assembly primitives. So that this initial CROSS-R-SDP port can serve as a reference for later optimizations and equivalence proofs, it minimizes usage of such primitives.

Sec. 4.1 gives a high-level overview of the separate parts involved in CROSS-Jasmin. Sec. 4.2 explains the design choices on the source-code level in detail and compares the development of CROSS-Jasmin to what a C programmer might expect. It also explains limitations of the Jasmin programming language relevant to CROSS, trade-offs made for the Jasmin port, and their impacts. Sec. 4.3 describes the automated tests and benchmarks for CROSS-Jasmin. Sec. 4.4 details the constant-time verification of CROSS-Jasmin, and sec. 4.5 formal proofs based on Jasmin's EasyCrypt extraction. Finally, sec. 4.6 summarizes the main limitations of the Jasmin toolchain relevant to the project.

4.1 Overview

The Jasmin port of CROSS has the following components, whose relationship is shown in fig. 4.1.1:

4.1.1 Fully functional Jasmin-language implementation of CROSS

The functionality of CROSS-R-SDP fast is implemented in the Jasmin language and compiled to x86_64 assembly. This implementation serves as the basis of our Jasmin port, can be linked to other software, be functionally- and constant-time verified using the toolchain, or used as a reference for performance optimizations. The Jasmin implementation covers all the features of the CROSS signature scheme to avoid off-loading functionality to a non-verified system, where high-assurance properties like cryptographic constant-time might be compromised. Just like the proof-of-concept library, our interface consists of the three procedures `KeyGen()`, `Sign()`, and `Verify()`.

4.1.2 Jasmin-language implementation of SHA-3

CROSS uses SHAKE [29] for its CSPRNG and as a hash function primitive. The proof-of-concept implementation can be configured to either use a public-domain, pure C implementation of SHAKE, or be linked against a system-installed library. This project requires a pure Jasmin implementation of SHAKE because external libraries can compromise security, and at worst nullify any assurance gained from using Jasmin. Re-using the predominant Jasmin SHA-3 library [6] could save time, but given that its interface is not easily compatible with CROSS, and implementing a cryptographic primitive is a realistic use-case of Jasmin that should be evaluated, we re-implement SHAKE in Jasmin.

4.1.3 C library-wrapper

A minimal C library that links against assembly produced from the Jasmin source code, manages random seeds, and allocates memory. This wrapper can be distributed for use in other C or C++ projects. To avoid introducing side-channel leakage, the library-wrapper contains as little logic as possible, and does not dereference any secret input- or output data.

4.1.4 Test bench and benchmarks

A test bench is implemented in C using the Unity [23] framework and checks correct functionality of CROSS-Jasmin with unit-tests and KAT generation. Additional benchmarks allow for comparing the code size, maximum stack usage, and runtime performance of CROSS-Jasmin and CROSS-C for a quantitative evaluation.

4.1.5 Constant-time verification

Machine-checked verification of CROSS-Jasmin's source code assures that the Jasmin compiler produces side-channel resistant assembly where the execution time does not depend on secret inputs. Verifying cryptographic constant-time of CROSS-Jasmin is non-trivial using either

the EasyCrypt extraction or the constant-time checker, because CROSS performs *probabilistic sampling* and branches on the hash of secret inputs. After evaluating both approaches, the constant-time checker was chosen to verify CROSS-Jasmin. Additionally, an example of verifying cryptographic constant-time based on the EasyCrypt leakage trace is given as a proof-of-concept for comparison.

4.1.6 EasyCrypt extraction and proofs

Beyond proving side-channel resistance, EasyCrypt enables verifying functional-correctness and security guarantees of Jasmin code (ch. 2). Some simple correctness properties of the CROSS port are stated as EasyCyrpt lemmas and proven to outline the process and limitations in this work, and to enable future work on the topic.

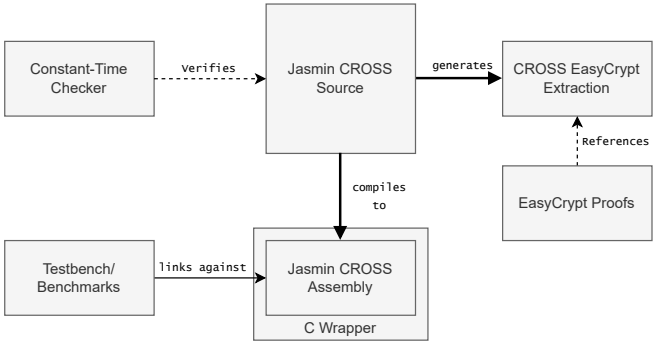


Figure 4.1.1 Relation between components of the CROSS signature scheme's Jasmin implementation.

4.2 Jasmin Program

Writing source code for CROSS-R-SDP fast in Jasmin shows that while sharing a lot of traits with high-level languages like C, it has notable differences and shortcomings at the language-level. The ability to define and re-use functions allows for a familiar programming style, which decomposes programs into libraries and avoid duplication by abstracting shared functionality as procedures. On the flip side, explicit register allocation, assembly primitives, and manual variable spilling expose instruction-set details that prevent many desired abstractions and force all parts of a program to keep the underlying processor in mind.

4.2.1 Cryptographic primitives

CROSS requires SHAKE to implement a cryptographically secure CSPRNG and hash function. Implementing the Keccak permutation in Jasmin is straightforward, and follows the description of the official standard [29]. The permutation is implemented as a function in Jasmin, which

receives a `reg mut ptr u64[25]` state, permutes it, and returns the modified array. Since Jasmin does not spill registers implicitly, a caller of the function has to ensure enough registers are available for intermediate results. For example, the code currently uses 7 registers for the permutation function, which leads to a compilation error due to register pressure when called from a context with a lot of allocated registers.

The Keccak permutation is called by the SHAKE hash function. SHAKE is implemented as both a non-incremental, and an incremental version used as a CSPRNG. Because Jasmin programs cannot modify global memory and function calls don't have side-effects, the incremental hash function has to return the modified state, and the caller has to explicitly pass it between calls. To implement CROSS' hash function in Jasmin, the incremental SHAKE implementation is wrapped with a non-incremental, fixed-length version. Listing 4.2.1 (a) shows the fixed-length hash function used by CROSS. The 'aux' variable holds auxiliary data that is required for incremental operation, such as the number of absorbed bytes. The `state_ptr` is explicitly returned by functions that modify it, due to Jasmin's functional array semantics (ch. 3).

Listing 4.2.1 (b) displays another shortcoming compared to CROSS-C: In C, it is possible to reference data by creating pointers to data objects, like arrays, and dynamically modify them and the values they reference. The Jasmin language currently has no way to generate pointers that reference stack data, all memory pointers need to be passed to Jasmin functions by the external environment. The CROSS signature scheme needs to hash differently sized, temporary, stack data and generate different length digests. In CROSS-C, there is a single SHAKE implementation that receives two data pointers with dynamic length parameters, but due to Jasmin's limitation, we need to duplicate the function for every fixed input- and digest lengths used in CROSS. While we do not have to duplicate source code because of Jasmin's parameterizable namespaces, the compiler generates duplicate assembly for each namespace, considerably increasing the binaries' size.

4.2.2 Arithmetic primitives

Other than the cryptographic primitives described in the previous section, most basic operations of CROSS are arithmetic, logical primitives. This includes computing matrix-vector products, subtracting, adding, or multiplying vectors, bit- packing and unpacking vectors, etc. The reference proof-of-concept implementation maps these operations to C procedures using simple while-loops. Such procedures translate well to Jasmin functions, with some additional boilerplate code for handling registers and temporary values. Jasmin gives developers the ability to implement loops using either backwards-branching while-, or unrolled for-loops, requiring users to make an explicit choice for each loop, a process that is instead handled automatically by optimizing C compilers. Almost all while-loops in the proof-of-concept representation could be replaced by for-loops that get unrolled, since the iteration count is fixed. Because loop-unrolling has unclear performance impacts, for example by increasing code size, increasing the rate of cache misses, or decreasing pipeline hazards, there is no simple answer on which loop version to use for each iteration in CROSS. For this reference implementation, loops with high iteration counts (ca. > 100) are implemented with while-loops, and for-loops are used for small iteration counts.

Passing data efficiently to- and from arithmetic primitives is currently a major challenge in Jasmin. Vectors and matrices have to be represented as fixed-size functional arrays, which

```
fn shake256_fixed(
    reg mut ptr u8[KC_DIGEST_SIZE]
      digest_out,
    reg ptr u8[KC_INPUT_SIZE] arr)
      -> reg mut ptr u8[KC_DIGEST_SIZE] {
  () = #spill(digest_out);
  stack u64[keccak_state_size] state;
  stack u64[keccak_aux_size] aux;
  reg mut ptr u64[keccak_state_size]
    state_ptr;
  state_ptr = state;
  state_ptr, aux =
    init_shake256(state_ptr, aux, arr);
  () = #unspill(digest_out);
  state_ptr, aux, digest_out =
    keccak_inc_squeeze(state_ptr, aux,
      digest_out);
  return digest_out;
}
```

```
namespace Keccak_384 {
  param int KC_INPUT_SIZE = 384;
  param int KC_DIGEST_SIZE = 48;
  require "keccak_fixed.jinc"
}
namespace Keccak_fq_vec_beta {
  param int KC_INPUT_SIZE =
      HASH_DIGEST_LENGTH;
  param int KC_DIGEST_SIZE =
      BYTES_BETA_ZQSTAR;
  require "keccak_fixed.jinc"
}
```

Listing 4.2.1 (a) Jasmin source code for the fixed-length CROSS hash function and (b) example namespaces.

cannot be referenced by pointers. Also, arrays cannot be turned into sub-arrays, or "slices", from run-time indices. One consequence is, as with the cryptographic primitives, that procedures for different vector lengths have to be duplicated, even when they perform the same computation. In addition, data often has to be redundantly copied between arrays: When a loop from the CROSS specification is implemented as a while-loop in Jasmin, so that it isn't being unrolled, the loop index is a run-time value, and thus cannot be used to slice arrays. In that case, the only option when calling a function that operates on a fixed-length slice, is to copy all contents of the sub-array into a temporary stack array, and pass it to the function. As an example, the 'Sign' procedure of CROSS calculates $y_i \leftarrow u'_i + beta[i] \cdot e'_i$ [9, Algorithm 2, line 26]. When using a while-loop in Jasmin, it isn't possible to address the slices e'_i, y_i, or u'_i, and pass them to a vector addition function, without copying the data.

While Jasmin has these shortcomings with data-sharing and code re-use, it handles bit-manipulation very well. Many arithmetic primitives operate on the bit-level, for example when packing vectors. Numerical data types are all unsigned integers, and thus naturally represent bit-vectors. All common logical operators, like bit-shifting, the boolean operators $\wedge$ $\vee$ $\oplus$ $\neg$, or rotations are defined as operators by the language. Further, assembly primitives can be used to specify exact register widths of arguments and results, and to assign flags to `reg bool` variables. Arrays can be interpreted as storing any numerical data types, no matter the declared type of the array, making it straightforward to reinterpret memory.

4.2.3 Algorithms and data-structures

At the conceptual level, CROSS uses more involved algorithms and data structures, which make use of the described primitives, to implement the signature scheme. These are, for

example, the Sign, Verify, and KeyGen procedures themselves, but also some lower-level functions, such as random sampling, seed expansion, or seed-tree generation. What differentiates these higher-level procedures from cryptographic- or arithmetic primitives is that they are more algorithmically involved: They may have multiple nested loops with dynamic iteration counts, may call different functions based on run-time values, and typically operate on more complicated data-structures, while aforementioned primitives operate on simple, linear byte sequences. These functions are more difficult to efficiently implement in Jasmin.

One problem that arises when handling data-structures is their representation in the programming language: The only aggregate data-type Jasmin supports are arrays of integers. Structs and multidimensional arrays from CROSS-C have to be flattened to simple byte arrays in CROSS-Jasmin, complicating code that operates on them. Operations on nested data-types are also often not possible to map to Jasmin because of the inability to take run-time references or to address subarrays, described in sec. 4.2.1 and 4.2.2.

Having to explicitly allocate and spill registers complicates writing algorithms that require many temporary values, indices, or counters. While not limiting a program's functionality as much as Jasmin's missing support for memory references does, manually handling register-allocation requires more resources to implement, and makes the source code less flexible and reusable. Functions are not as good of an abstraction in Jasmin, as calling them can result in register allocation errors if not enough registers are free at the callsite. Since register allocation is an ISA-dependent process, it exposes details of the underlying machine to developers and makes source code architecture dependent, even if no primitive assembly instructions are used. For example, arguments to a function have to be allocated according to the ABI's calling convention, which results in a compile-time error if that specific register is required by an operation in the function's body. Handling stack- vs. register-storage is further complicated by the restriction of certain operations to certain storage classes, according to the ISA. If the CPU only supports register operands to an instruction, then Jasmin programs have to use register-allocated variables. To illustrate, on x86 the bit-shift instructions requires the second argument to be in register `CL`, so the Jasmin program `stack u64 x y; x << y` cannot be compiled.

4.3 Automated Tests

Writing automated unit- or integration tests for Jasmin code can be done in any language that links with C object files. For this project, the simple C testing framework Unity [23] was used. Testing is not only used to ensure correct functionality, it is also necessary for debugging Jasmin programs during development. Since Jasmin has no option for I/O and the compiler doesn't support generation of debugging symbols, users are left with testing input-output behavior of exported functions. The simplest approach to debugging is to split a faulty function into smaller, exported units, and to test their outputs separately until the source of error is found. Interactive debugging of Jasmin code is possible to the degree of single-stepping through assembly instructions, which may be helpful for simple, short functions.

The test suite for CROSS' Jasmin implementation is comprised of:

- Unit tests of individual functions: Arithmetic-, and cryptographic primitives are tested according to known input and results.

- Integration tests of the Sign, Verify, and KeyGen functions: The end-to-end behavior of CROSS is tested according to sample keys, signatures, and messages generated by the proof-of-concept C implementation.
- Automated Known-Answer-Tests: The test vectors from the known-answer-tests of CROSS are produced and verified according to the published data.

Because of the strict Jasmin memory model, and the absence of pointers, most sources of error were due to faulty logic, and not because of unsafe memory accesses. Developing a test suite for Jasmin programs is vital to find- and resolve programming errors.

4.4 Cryptographic Constant-time Verification

The constant-time verification of CROSS-Jasmin is done using the constant-time checker (sec. 3.3). A number of functions have also been verified using the EasyCrypt extraction (sec. 3.2) to compare both approaches.

4.4.1 Verification using EasyCrypt extraction

The initial approach for ensuring constant-time execution of CROSS was to show leakage resistance via the EasyCrypt extraction. The formalization of cryptographic constant-time in EasyCrypt was added to Jasmin before the constant-time checker, and most existing guides and literature refer to it rather than using the security type system. Verifying the absence of leakage is straightforward for CROSS' cryptographic- and arithmetic primitives, whose execution path is completely oblivious to secret inputs. For example, listing 3.2.1 shows the specification and proof script for proving that the runtime and memory access pattern of CROSS-Jasmin's `shake256()` primitive only depends on the length of the input and the pointer values of array arguments.

However, verifying constant-time security using this method is tedious and inflexible once random sampling and one-way functions are introduced to a program. Jasmin instruments the EasyCrypt extraction with a global leakage trace that accumulates all branch conditions and memory addresses. EasyCrypt can use automated tactics to show the leakage trace is always the same, as was done for shake256, but it can't reason if a trace is security-critical or not (sec. 3.3). CROSS performs both branching on hashed secret values and random sampling from the CSPRNG initialized with a secret seed. Let's consider the `sim` tactic to show why the standard approach of proving PRHL equalities fails: `sim` works by working backwards from the PRHL judgment's postcondition, consuming statements from the program [19]. At each step it updates the equalities of the postcondition according to the consumed statement until it reaches the precondition. For example, to show that:
`={y} (x <- y) ~ (x <- y) ={x}` holds, `sim` propagates the postcondition through the assignment, and leaves the implication `={y} ==> ={y}` as the goal. Clearly, in case of random sampling or when using deterministic hash functions, the leakage trace still depends on the secret values, although they are not security critical.

To handle these cases in EasyCrypt, the user has to axiomatize security assumptions, for example that hash function outputs are independent of their input values, or develop a more complicated formalization of cryptographic constant-time and corresponding proofs. For this

project we focus on the former approach, while [5] uses the latter for random sampling. One instance of random sampling in CROSS is the generation of the private vector η, which is used to generate the error vector e. Because η is defined over the finite field $\mathbb{F}_7$ in CROSS-R-SDP, some random values from the CSPRNG have to be discarded, and so the runtime depends on the actual sequence of values generated from the secret seed. However, because the CSPRNG is non-invertible, leaking the number of discarded values does not compromise the secret, and the discarded values of η are never used during signature generation. Listing A.1 shows the EasyCrypt specification and proof of the function sampling η. We add an axiom stating that SHAKE is equivalent to sampling a completely random value to break the dependence between hash digest and input. We can then write a more involved proof script with Hoare logic tactics making use of that fact to show that the function still produces the same leakage trace. Note that adding axioms can introduce errors and allow users to proof false lemmas. For example, the axiom of listing A.1 removes the dependence of hash inputs and digests, meaning that η could now be considered public if the same formalization is used, even though it is a secret value of the scheme. To avoid mistakes like this, care needs to be taken when writing axioms and interpreting logical specifications.

4.4.2 Verification using the constant-time checker

[3] Used Jasmin's security type system, and constant-time checker for verifying that Kyber [14] is leakage-free. To deal with probabilistic sampling and cryptographic hash functions, they use the type system's *declassify* mechanism (sec. 3.3). For CROSS, the security type system makes constant-time verification much simpler and easier to maintain than EasyCrypt proof scripts. It also requires less computational resources to verify security types than to evaluate a proof script with EasyCrypt using costly smt tactics. The constant-time checker supports type inference for all non-annotated variables, which means that only the secret inputs and outputs of CROSS have to be annotated. The checker ensures that no secret value gets propagated to a public variable, and that addresses and branch conditions only depend on public variables.

The two exported functions of CROSS-Jasmin that must be annotated with types are Sign and KeyGen, while the Verify function does not receive any secret data. Listing 4.4.1 shows the signatures of both functions as present in the source code. Classifying the seed parameter of the KeyGen procedure ensures no data about the generated secret key can leak, because the key is generated solely from the seed, and its type 'secret' gets propagated through the program. In total, CROSS-Jasmin requires eight declassifications: One in each of the four random sampling functions, two to declassify the part of the CSPRNG state that tracks the number of absorbed bytes, one for the digest d_b, which is generated from secret values, and one to declassify the public key generated from the private key. The `sample_vec()` function of listing 4.4.1 gives an example of how declassification is used.

One issue arose with the incremental version of SHAKE. The incremental implementation usually extends the state array of SHAKE by one additional element, counting the number of absorbed bytes. Conceptually, the state of a hash function should be a secret value, but the additional counter has to be public, since SHAKE's execution time depends on it. Jasmin does not allow for individual cells of arrays to have different security types. To avoid unnecessarily declassifying the entire state, the additional count is passed explicitly as an auxiliary parameter. This way the state can be annotated with 'secret', while the auxiliary parameter can be 'public'.

```
export fn cross_sign(... ,
    #[secret] reg mut ptr u64[26] platform_seed_, ...,
    #[secret] reg ptr u8[KEYPAIR_SEED_LENGTH_BYTES] skey)
  -> reg mut ptr u8[SIG_LENGTH], reg mut ptr u64[26]
export fn cross_keygen(
    #[secret] reg mut ptr u8[KEYPAIR_SEED_LENGTH_BYTES] sk_,...
    #[secret] reg mut ptr u64[26] platform_seed_)
  -> ...
fn sample_vec(...) -> ... {
  ...
  #[declassify] tmp = tmp;
  if(tmp < Q) {
      placed += 1;
      sub_buffer = sub_buffer >> bits_for_q;
      bits_in_sub_buf -= bits_for_q;
  } else {
      sub_buffer = sub_buffer >> 1;
      bits_in_sub_buf -= 1;
  }
  ...
}
```

Listing 4.4.1 CROSS-Jasmin security type annotations and declassification.

Of course, errors are introduced when declassifying data that should not actually be considered public knowledge. The reason we can declassify these variables in CROSS is because (i) they are public values of the scheme, generated by cryptographically secure functions from secret data, or (ii) the randomly sampled values that influence the execution time are discarded.

4.5 Functional Correctness Properties

EasyCrypt's program logic enables verifying *functional correctness* of extracted Jasmin code [1]. Functional correctness, in our case, is the property that our program's functionality fulfills some specification. Specifications have different levels of detail, for example, specifying that a function never produces negative values, or specifying that it always produces exactly the same value as some reference model. Because the verification of complex software is very time-, and resource-intensive [24, 25], this project limits itself to a few, simple functional correctness properties. The goal of these is not to increase the assurance of CROSS-Jasmin, but to further evaluate the Jasmin toolchain.

Functional correctness proofs for Jasmin programs are well-supported because of Jasmin's integration with EasyCrypt. Namely, EasyCrypt's "weakest precondition" tactic, and its integration with SMT-solvers can automate a lot of otherwise labor-intensive proof steps. Listing A.2 outlines a simple functional correctness proof of a CROSS function used by the Verify procedure, which checks if a vector is in the expected field. The lemma states that if all elements are in the field (< 7), the function returns 1. Note that most proof steps are dispatched by `smt` and `wp` tactics, limiting the main manual step to providing the loop invariant via `while`.

4.6 Limitations

This section summarizes the main limitations encountered during the development of CROSS-Jasmin. Table 4.1 references the relevant sections for the limitation and the component of the Jasmin toolchain it is caused by. *Lack of memory references* is a language-level limitation of Jasmin, making it impossible to pass dynamically-sized arrays. The limitation reduces code reusability and may necessitate redundant copies of data. *Lack of nested data types* describes both the inability to user-define nested data types, such as structs, and the lack of multi-dimensional arrays. *Need for manual register- and stack allocation* is a trade-off between the decrease in abstraction, and the ability for programmers to carefully decide in which exact register class a value will reside throughout an execution. *No debugging symbols* is related to the compiler, which currently cannot generate debugging symbols for Jasmin source code. Together with a lack of printing values from Jasmin, it reduces the debugging methodology mostly to unit-tests. *Few automated performance optimizations* results from the deliberate design choice to have the compiler produce *predictable* assembly: The final assembly should perform instructions as a programmer would expect, given the source code. The level of compile-time optimization is also restricted due to the constant-time guarantees made by Jasmin. As a consequence, many transformations that an optimizing C compiler would typically do now have to be explicitly done at the source code level, for example by using vector instruction primitives. Finally, *No EasyCrypt leakage model for declassification* describes that the leakage trace model used by Jasmin is not easily compatible with programs that need to declassify values, for example due to probabilistic sampling in CROSS, or the computation of a public- from a private-key. Instead, users would have to write considerably more involved proofs and specifications.

Table 4.1 Relevant Jasmin limitations for the implementation of CROSS.

Limitation	Component	Section
Lack of memory references	Jasmin Language	4.2.1, 4.2.2
Lack of nested data types	Jasmin Language	4.2.3
Need for manual register- and stack allocation	Jasmin Language	4.2.3
No debugging symbols	Jasmin Compiler	4.3
Few automated performance optimizations	Jasmin Compiler	5.1
No EasyCrypt leakage model for declassification	Jasmin Compiler	4.4

5 Evaluation

This chapter provides a quantitative and qualitative evaluation of the Jasmin toolchain. Sec. 5.1 analyzes performance metrics of generated code, and sec. 5.2 discusses advantages and disadvantages of utilizing Jasmin that were observed during implementation of CROSS, and concludes by describing a number of alternative tools for high-assurance cryptographic software.

5.1 Measurements

Fig. 5.1.1 shows the mean execution time of cryptographic routines of CROSS-C and CROSS-Jasmin, the bars are annotated with $\frac{t}{100ms}$ for execution time t. The reference Jasmin implementation is slower than the proof-of-concept C implementation by a factor of ca. 5. Due to the limited number of optimizations the Jasmin compiler can perform, these values are in the expected range. The maximum stack usage and binary size are compared in fig. 5.1.2. Note that neither CROSS-C nor CROSS-Jasmin allocates memory dynamically, so their stack usage is representative of total memory usage at runtime. The memory usage of CROSS-Jasmin is nearly the same as that of CROSS-C. Meanwhile, the binary sizes differ by a factor of ca. 4, which is due to the need for code repetition and unrolling in Jasmin explained in sec. 4.6.

5.1.1 Methodology

The variant used for both CROSS-C and CROSS-Jasmin in all measurements is *CROSS CAT 3 R-SDP fast*. The evaluation-platform is a Linux machine with a x86_64 processor. The platform's configuration is summarized in table 5.1. The speed metrics are given as a statistical mean to exclude deviation caused by the operating system, such as scheduling decisions. The size metrics are deterministic for any given arguments to the measured functions and thus don't require statistical analysis. The Sign and Verify procedures were invoked with a 32-bytes long message.

Table 5.1 Evaluation-platform's configuration.

Component	Name	Version
CPU	AMD Ryzen 9 7950X	Zen 4
OS	Linux	6.12.3
C-Compiler	GCC	13.3.0
Jasmin-Compiler	jasminc	v2024.07.0 (e4640e7)

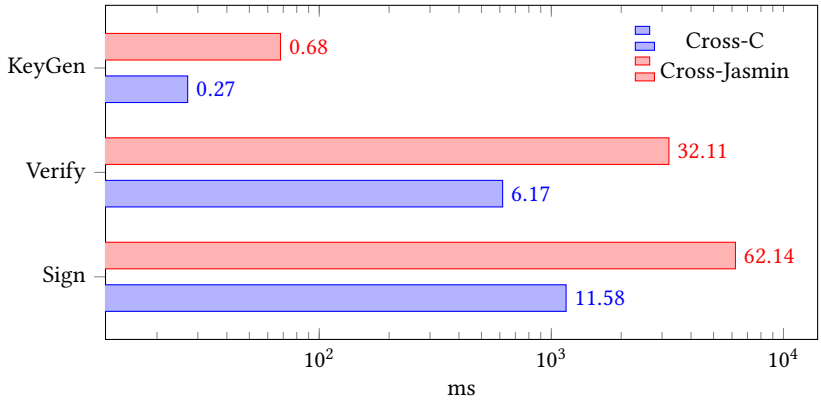


Figure 5.1.1 CROSS speed performance measurements of the Proof-of-concept C implementation vs. the Jasmin implementation.

5.2 Discussion

First, we discuss the benchmarking results from the previous section. Both the proof-of-concept C implementation, and CROSS-Jasmin follow an "idiomatic" coding style, and do not attempt to optimize the programs for performance, such as by using vector instructions. We expect the lower speed of CROSS-Jasmin to be due to limitations imposed on the compiler, which prohibit multiple kinds of compiler-optimizations, such as dead code elimination. While the Jasmin compiler has to maintain cryptographic constant-time and use exact registers and memory operations as defined in the source code, an optimizing C compiler only has to produce the correct result on procedure exit. Writing optimized Jasmin programs by using, for example, vector intrinsics, and minimizing memory pressure, would likely close the gap between the C- and Jasmin-implementation's performance. [6, p. 1621] demonstrate that their optimized AVX2 implementation of SHA-3 in Jasmin outperforms OpenSSL's version. It would be more difficult to reduce the binary size of CROSS-Jasmin to an amount comparable with CROSS-C. As detailed in sec. 4.6, the Jasmin language currently has limitations, which require code duplication for different argument sizes or compile-time parameters. This issue would best be solved with extensions to the language. Another likely cause of increased code size is that all for-loops get implicitly unrolled by the compiler. It is possible to replace every for-loop with a while-loop, although the user has to carefully examine the size- and performance impact of using branching- vs. unrolled code for each loop. A C compiler instead uses heuristics to decide on which loops to unroll, and how many times to unroll it. It would also be possible to extend the Jasmin compiler in the future with similar heuristics. As a first, non-optimized reference implementation, the Jasmin program's performance is promising when considering that it is verifiably cryptographic constant-time.

Second is a discussion of the development of Jasmin-C and accompanying EasyCrypt proofs. This part emphasizes the major shortcomings of Jasmin compared to a typical C implementation that could limit future work with the toolchain. Constraining the whole development process

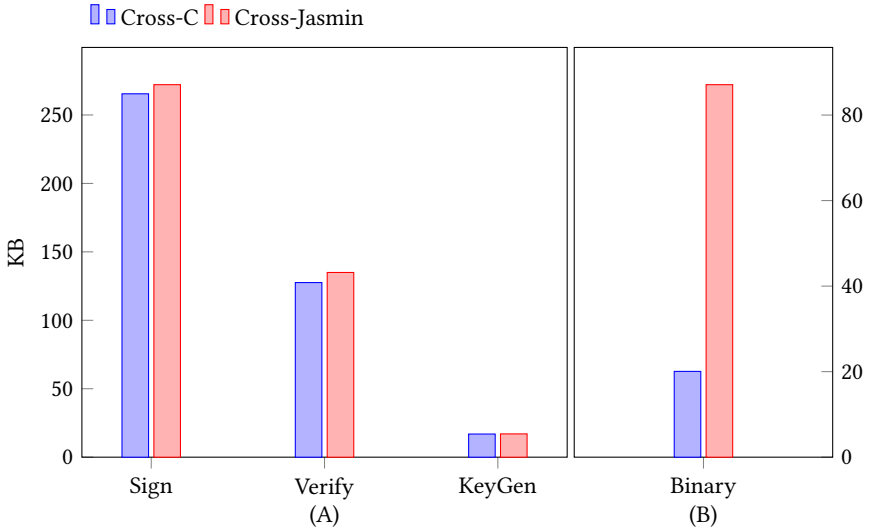


Figure 5.1.2 CROSS size measurements of the Proof-of-concept C implementation vs. the Jasmin implementation. (A) Stack usage; (B) Flash usage (Text + Data + BSS).

with Jasmin, regardless of the specific project, is that it takes considerably more resources than typical, high-level languages require. Because a programmer has to explicitly manage registers, stack slots, spilling, and manually perform optimizations, it is close to an "Assembly-level" programming model. This feature of the Jasmin language has both the negative effect of shifting tasks from the compiler to users, which they have to account for, and the positive effect of giving them a large level of control, which might be necessary to avoid side-channel leakage. [4] call this aspect *predictability*, because the programmer can predict what assembly will be generated from source code. A major issue during the development of CROSS-Jasmin were missing constructs for memory references from the Jasmin-language, which result in being unable to reuse functions with different array sizes and impose unnecessary copies of data onto the program. This proved particularly challenging for implementing arithmetic and cryptographic primitives for CROSS, which are often reused for multiple in- and output lengths throughout the program. This limitation is not necessary for cryptographic constant-time and could be resolved in the future by adding features to the language. Lastly, we want to point out that Jasmin is not well-documented, and some features are only described in the commit history, or not described at all. For example, the fact that arrays can be divided, or "sliced" is not mentioned in the official documentation. Multiple points throughout development required reading the compiler's source-code to learn about Jasmin language constructs.

Proving functional correctness or safety properties about Jasmin programs is relatively simple thanks to the interoperability with EasyCrypt. EasyCrypt has a flexible system for adding theories and has a set of automated tactics to reason about imperative programs as well as higher-order logic. Its integration with SMT solvers means that most program proofs can

be carried out by providing correct loop invariants, calculating a verification condition, and dispatching it with automated solvers. The overhead for an experienced user is reasonable, especially because cryptographic schemes mostly use simple algorithms and data-structures. Two issues that arose related to verifying the EasyCrypt extraction of CROSS-Jasmin were (i) that the leakage-model is insufficient for proving random sampling correct, and (ii) that refactoring Jasmin code requires existing proofs to be adapted, because they are written with the code's structure in mind. Luckily, issue (i) could be avoided by using the constant-time checker instead, while (ii) has to be explicitly accounted for by development practices.

Finally, we want to reference alternative tools to Jasmin for high-assurance cryptographic implementation. Some authors, such as [7] have proved functional correctness of a scheme implemented in C and then used a verified compiler, such as CompCert [26], to ensure properties are maintained by the binary. However, while CompCert produces functionally correct machine-code, it does not make guarantees about execution-time side-channel leakage. Vale [13] is a tool for verifying cryptographic code at the assembly-level. Vale is a programming language, similar to Jasmin, which gets compiled into an architecture-specific abstract syntax tree (AST) in Dafny [22]. Dafny is then used to verify properties about the code, such as functional-correctness or side-channel resistance, and outputs the final assembly code. Qhasm [11] is a programming language for cryptographic software that combines the precision of assembly-level programming with generic, machine-independent operators, such as addition or bit-shifting. Jasmin is inspired by qhasm, but introduces more high-level language concepts like loops and functions [4]. Qhasm allows for machine-independent development of high-speed cryptographic software as opposed to writing direct assembly, but it does not have a formally verified semantics or a verification toolchain, which makes it unsuitable for high-assurance, verified programs. Also similar to Jasmin is ct-verif [2], which verifies constant-time security of code on the LLVM intermediate representation. Because the verification is performed close to final machine-code generation, the authors argue that constant-time security gets preserved. Unlike in Jasmin, this property could still be violated by compiler optimization of the verified code.

6 Outlook

This work's contributions, namely a fully functional, cryptographic constant-time, reference implementation of CROSS in Jasmin and the related evaluation of its toolchain and EasyCrypt are a foundation for potential future topics. This chapter describes selected future work and what their main contributions would be.

Extend CROSS-Jasmin to multiple variants

CROSS-Jasmin currently implements CROSS-R-SDP fast. Interesting extensions would be CROSS-R-SDP(G), which bases its cryptography on slightly different arithmetic for performance reasons, and CROSS 'balanced', which uses different high-level algorithms to achieve smaller signature sizes [9]. For instance, CROSS-Jasmin currently has a signature size of ca. 43 KB, updating the implementation to add support for R-SDP(G) would decrease the size to ca. 27 KB while maintaining the same security level. The main benefit would be a more efficient, constant-time CROSS implementation in Jasmin. The algorithmic changes are limited and mostly reduce to updating the arithmetic primitives. In comparison, adding support for the 'balanced' variant would decrease signature sizes to ca. 28 KB. CROSS-balanced uses different data structures and algorithms during operation, and could thus be an interesting new use-case for Jasmin other than offering a performance improvement.

Optimize CROSS-Jasmin

The current implementation in Jasmin is a reference version, and is written with few optimizations. Jasmin is designed with manual source-code level optimizations in mind, similar to an assembly-level language. For example, a new version of CROSS could replace arithmetic- and cryptographic primitives with vectorized version by using direct machine-level intrinsics (sec. 3.1.4). An orthogonal approach for optimization is to remove redundancies in existing CROSS-Jasmin code, for example eliminating redundant copies of arrays or using better data structures for intermediate results. Such work would contribute, again, not only a more efficient, high-assurance implementation of CROSS, but also evaluate Jasmin for high-performance cryptography.

Further verification of CROSS-Jasmin

EasyCrypt allows to verify both the security and functional correctness of Jasmin programs. [3], for example, verify security properties of Kyber and show that their Jasmin implementation is functionally correct with respect to a specification in EasyCrypt, and [6] follow a similar approach for SHA-3. Similarly, CROSS could be specified as a model in EasyCrypt, proven secure, and then related to a Jasmin implementation, such that the compiled code is asserted to implement the abstract model. This would require considerable effort to get accustomed to verification methods of security protocols and imperative programs, to specify CROSS in

EasyCrypt, and to develop all proofs and intermediary steps necessary to complete a proof. The main contributions would naturally be a much more trustworthy implementation of CROSS, as well as a formal proof of its security.

Extension of Jasmin

Finally, Jasmin could be extended to address the limitations described in sec. 4.6 by adding features to the language and writing accompanying proofs for the compiler. Independent of language features, the Jasmin compiler could be extended with support for new architectures, such as RISC-V [34]. This work would make Jasmin more practical for future use by making code easier to write, and have Jasmin produce more efficient assembly. Any work on the Jasmin compiler or language also necessitates writing new proofs in Coq that formalize the language semantics and show that the compiler preserves them.

List of Figures

List of Tables

List of Listings

Acronyms

AST abstract syntax tree

CROSS codes and Restricted Objects Signature Scheme

CROSS-C proof-of-concept CROSS implementation in C

CROSS-Jasmin this work's reference CROSS implementation in Jasmin

CSPRNG cryptographically secure pseudorandom number generator

HL Hoare logic

PHL probabilistic Hoare logic

PRHL probabilistic relational Hoare logic

References

[1] Jose Bacelar Almeida et al. “The Last Mile: High-Assurance and High-Speed Cryptographic Implementations”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2020.

[2] Jose Bacelar Almeida et al. “Verifying Constant-Time Implementations”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016.

[3] José Bacelar Almeida et al. “Formally Verifying Kyber: Episode V: Machine-Checked IND-CCA Security and Correctness of ML-KEM in EasyCrypt”. In: *Advances in Cryptology – CRYPTO 2024*. Springer Nature Switzerland, 2024, pp. 384–421.

[4] José Bacelar Almeida et al. “Jasmin: High-Assurance and High-Speed Cryptography”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. ACM, Oct. 2017.

[5] José Bacelar Almeida et al. *Leakage-Free Probabilistic Jasmin Programs*. Cryptology ePrint Archive, Paper 2023/1514. Version 20231006:122313. Oct. 6, 2023. https://eprint.iacr.org/2023/1514/20231006:122313.

[6] José Bacelar Almeida et al. “Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’19. ACM, Nov. 2019.

[7] Andrew W. Appel. “Verification of a Cryptographic Primitive: SHA-256”. In: *ACM Transactions on Programming Languages and Systems* 37.2 (Apr. 2015).

[8] Marco Baldi et al. *CROSS Implementation*. Aug. 13, 2024. https://github.com/CROSS-signature/CROSS-implementation (visited on 11/25/2024).

[9] Marco Baldi et al. *CROSS: Codes and Restricted Objects Signature Scheme*. Version 1.2. Feb. 3, 2024. https://www.cross-crypto.com/CROSS_Specification_v1.2.pdf (visited on 07/05/2024).

[10] Gilles Barthe et al. “Computer-Aided Security Proofs for the Working Cryptographer”. In: *Advances in Cryptology – CRYPTO 2011*. Springer Berlin Heidelberg, 2011, pp. 71–90.

[11] Daniel Bernstein. *qhasm*. 2005. http://cr.yp.to/qhasm.html (visited on 11/18/2024).

[12] Daniel J. Bernstein and Tanja Lange. “Post-quantum cryptography”. In: *Nature* 549.7671 (Sept. 2017).

[13] Barry Bond et al. “Vale: Verifying High-Performance Cryptographic Assembly Code”. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017.

[14] Joppe Bos et al. *CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM*. Cryptology ePrint Archive, Paper 2017/634. Version 20201014:095148. Oct. 14, 2020. https://eprint.iacr.org/2017/634/20201014:095148.

[15] CVC5 Contributors. *CVC5*. 2024. https://cvc5.github.io/ (visited on 10/09/2024).

[16] Denis Firsov and Dominique Unruh. “Zero-Knowledge in EasyCrypt”. In: *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. Vol. 1. IEEE, July 2023.

[17] Formosa Crypto. *Libjade*. 2024. https://github.com/formosa-crypto/libjade/tree/main (visited on 11/19/2024).

[18] Formosa Crypto and EasyCrypt Contributors. *EasyCrypt*. 2024. https://www.easycrypt.info/ (visited on 10/09/2024).

[19] Formosa Crypto and EasyCrypt Contributors. *EasyCrypt Reference Manual*. Sept. 27, 2024. https://www.easycrypt.info/easycrypt-doc/refman.pdf (visited on 10/09/2024).

[20] IMDEA Software Institute et al. *Jasmin*. 2024. https://github.com/jasmin-lang/jasmin (visited on 11/18/2024).

[21] Inria. *The Coq Proof Assistant*. July 5, 2024. https://coq.inria.fr/ (visited on 07/05/2024).

[22] Leino K. and Rustan M. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer Berlin Heidelberg, 2010, pp. 348–370.

[23] Mike Karlesky, Mark VanderVoord, and Greg Williams. *Unity*. Mar. 10, 2024. https://github.com/ThrowTheSwitch/Unity (visited on 11/25/2024).

[24] Gerwin Klein et al. "seL4: formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP09. ACM, Oct. 2009.

[25] Xavier Leroy. "A Formally Verified Compiler Back-end". In: *Journal of Automated Reasoning* 43.4 (Nov. 2009).

[26] Xavier Leroy. "Formal certification of a compiler back-end or: programming a compiler with a proof assistant". In: *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL06. ACM, Jan. 2006.

[27] Microsoft Corporation. *Z3 Theorem Prover*. 2024. https://github.com/Z3Prover/z3 (visited on 10/09/2024).

[28] National Institute of Standards and Technology. *Call for Additional Digital Signature Schemes forthe Post-Quantum Cryptography Standardization Process*. Online. Sept. 6, 2022. https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf (visited on 12/31/2024).

[29] National Institute of Standards and Technology. *SHA-3 standard: permutation-based hash and extendable-output functions*. 2015.

[30] Tobias Nipkow. "Structured Proofs in Isar/HOL". In: *Types for Proofs and Programs*. Springer Berlin Heidelberg, 2003, pp. 259–278.

[31] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer International Publishing, 2014.

[32] OpenSSL Software Services Inc. and OpenSSL Software Foundation Inc. *OpenSSL*. 2024. https://github.com/openssl/openssl/tree/master (visited on 12/16/2024).

[33] Benjamin C. Pierce et al. *Programming Language Foundations*. Ed. by Benjamin C. Pierce. Vol. 2. Software Foundations. Electronic textbook, 2024.

[34] RISC-V International. *RISC-V*. 2024. https://riscv.org/specifications/ratified/ (visited on 12/31/2024).

[35] Basavesh Ammanaghatta Shivakumar et al. "Typing High-Speed Cryptography against Spectre v1". In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2023.

[36] Victor Shoup. *Sequences of games: a tool for taming complexity in security proofs*. Cryptology ePrint Archive, Paper 2004/332. Version 20060118:151030. Jan. 18, 2006. https://eprint.iacr.org/2004/332/20060118:151030.

[37] Toccata. *Why3*. 2024. https://www.why3.org/ (visited on 11/01/2024).

[38] Yuval Yarom and Naomi Benger. *Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack*. Cryptology ePrint Archive, Paper 2014/140. Version 20140227:013343. Feb. 27, 2014. https://eprint.iacr.org/2014/140/20140227:013343.

A

```
op hash_distr: (W8.t Array92.t) distr.
axiom hash_distr_lossless: is_lossless hash_distr.
axiom hash_distr_funiform: is_funiform hash_distr.
module Permutation = {
  proc perfect_permutation(
    state_ptr: W64.t Array25.t,
    aux: W64.t Array1.t,
    digest_out: W8.t Array92.t) : W8.t Array92.t =
  {
    var result: W8.t Array92.t;
    result <$ hash_distr;
    return result;
  }
}.
axiom perfect_permutation:
equiv[Cross_ct.M.keccak_zz_vec__keccak_inc_squeeze
      ~ Permutation.perfect_permutation :
      ={leakages} ==> res.`3{1} = res{2} /\ ={leakages}].
equiv zz_vec: Cross_ct.M.zz_vec ~ Cross_ct.M.zz_vec:
  ={leakages} ==> ={leakages}.
proof.
  proc. seq 13 13 : (={aux_1, leakages, bits_for_z, mask}); first last.
  + inline *. sim.
  + call (_: ={leakages} ==> res.`3{1} = res.`3{2} /\ ={leakages}).
    + transitivity Permutation.perfect_permutation
  (={leakages} ==> res.`3{1} = res{2} /\ ={leakages})
  (={leakages} ==> res{1} = res.`3{2} /\ ={leakages});
      progress; auto.
      smt. apply perfect_permutation. symmetry.
      conseq (_: ={leakages} ==> res.`3{1} = res{2} /\ ={leakages});
      auto. apply perfect_permutation.
    + conseq (_: ={leakages} ==> ={leakages, bits_for_z, mask}) => //.
      progress. sim.
qed.
```

Listing A.1 Cryptographic constant time proof of probabilistic sampling in CROSS.

```
lemma zz_vec_in_restr_group_1:
    hoare[Cross.M.is_zz_vec_in_restr_group :
          (Array187.all(fun (v : W8.t) => W8.to_uint(v) < 7)(arg))
          ==> res = W8.of_int(1)].
    proc. wp.
    while (W64.zero \ule ctr /\
  ok = W8.one /\ ctr \ule (W64.of_int 187) /\
  Array187.all(fun (v : W8.t) => v \ult W8.of_int 7)(in_0)).
    wp. auto. progress. smt. rewrite Array187.allP in H1.
    have excl : in_0{hr}.[to_uint ctr{hr}] \ult (of_int 7)%W8.
    smt. smt. smt. wp. auto. progress. rewrite Array187.allP in H.
    rewrite Array187.allP. auto; by smt.
qed.
```

Listing A.2 Functional-correctness proof of arithmetic primitive in CROSS-Jasmin.

```
proc is_zz_vec_in_restr_group (in_0:W8.t Array187.t) : W8.t = {
  var aux_1: bool;
  var aux_0: W8.t;
  var aux: W64.t;

  var ok:W8.t;
  var ctr:W64.t;
  var b:bool;
  var tmp:W8.t;

  leakages <- LeakAddr([]) :: leakages;
  aux <- (W64.of_int 0);
  ctr <- aux;
  leakages <- LeakAddr([]) :: leakages;
  aux_0 <- (W8.of_int 1);
  ok <- aux_0;

  leakages <-
    LeakCond((ctr \ult (W64.of_int 187))) :: LeakAddr([]) :: leakages;
  while ((ctr \ult (W64.of_int 187))) {
    leakages <- LeakAddr([(W64.to_uint ctr)]) :: leakages;
    aux_1 <- (in_0.[(W64.to_uint ctr)] \ult (W8.of_int 7));
    b <- aux_1;
    leakages <- LeakAddr([]) :: leakages;
    aux_0 <- SETcc b;
    tmp <- aux_0;
    leakages <- LeakAddr([]) :: leakages;
    aux_0 <- (ok & tmp);
    ok <- aux_0;
    leakages <- LeakAddr([]) :: leakages;
    aux <- (ctr + (W64.of_int 1));
    ctr <- aux;
    leakages <-
      LeakCond((ctr \ult (W64.of_int 187))) :: LeakAddr([]) :: leakages;
  }
  return (ok);
}
```

Listing A.3 EasyCrypt extraction with leakage trace of CROSS-Jasmin arithmetic primitive.