Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust

Denis Merigoux, Franziskus Kiefer, Karthikeyan Bhargavan

hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust

Denis Merigoux
Inria
denis.merigoux@inria.fr

Franziskus Kiefer
Wire
mail@franziskuskiefer.de

Karthikeyan Bhargavan
Inria
karthikeyan.bhargavan@inria.fr

***Abstract*—Despite significant progress in the formal verification of security-critical components like cryptographic libraries and protocols, the secure integration of these components into larger unverified applications remains an open challenge. The first problem is that any memory safety bug or side-channel leak in the unverified code can nullify the security guarantees of the verified code. A second issue is that application developers may misunderstand the specification and assumptions of the verified code and so use it incorrectly.**

In this paper, we propose a novel verification framework that seeks to close these gaps for applications written in Rust. At the heart of this framework is hacspec, a new language for writing succinct, executable, formal specifications for cryptographic components. Syntactically, hacspec is a purely functional subset of Rust that aims to be readable by developers, cryptographers, and verification experts. An application developer can use hacspec to specify and prototype cryptographic components in Rust, and then replace this specification with a verified implementation before deployment. We present the hacspec language, its formal semantics and type system, and describe a translation from hacspec to F⋆. We evaluate the language and its toolchain on a library of popular cryptographic algorithms.

An earlier attempt in this direction by some of the same authors, was also called hacspec, and sought to embed a cryptographic specification language into Python [13]. We now believe that the strong typing of Rust provides an essential improvement to the specification and programming workflow. This work subsumes and obsoletes that earlier attempt. Hereafter, we use hacspec-python to refer to this prior version.

1. Introduction

Modern Web applications use sophisticated cryptographic constructions and protocols to protect sensitive user data that may be sent over the wire or stored at rest. However, the additional design complexity and performance cost of cryptography is only justified if it is implemented and used correctly. To prevent common software bugs like buffer overflows [40] without compromising on performance, developers of security-oriented applications, like the Zcash and Libra blockchains, are increasingly turning to strongly typed languages like Rust. However, these type systems cannot prevent deeper security flaws. Any side-channel leak [1] or mathematical bug [20] in the cryptographic library, any parsing bug [27] or state machine flaw [11] in the protocol code, or any misused cryptographic API [18] may allow an attacker to steal sensitive user data, bypassing all the cryptographic protections.

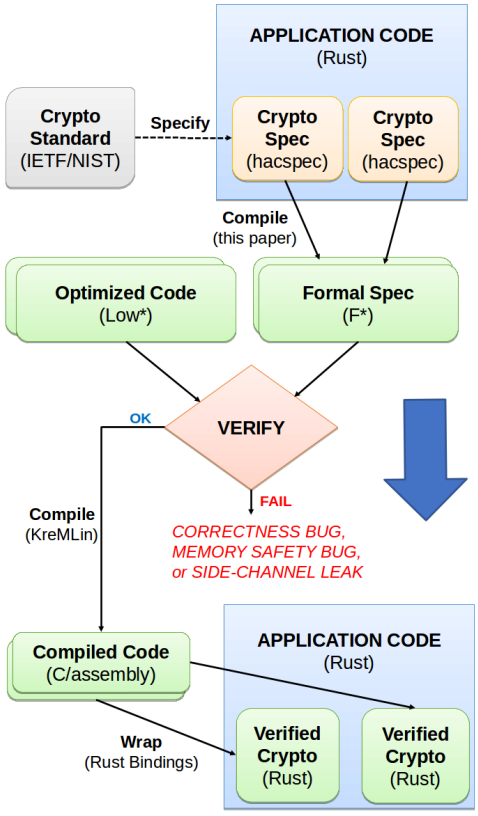

Figure 1. **hacspec programming and verification workflow.** The Rust programmer writes executable specifications for cryptographic components in hacspec and compiles them to formal specifications (in F⋆). The proof engineer implements the cryptographic components (in Low* or Jasmin) and proves that they meet their specifications. The verified code is compiled to high-performance C or assembly code, which is finally wrapped within Rust modules (using foreign function interfaces) that can safely replace the original hacspec specifications.

The problem is that these kinds of deep bugs often appear only in rarely-used corner-cases that are hard to find by random testing, but can be easily exploited by attackers who know of their existence. Furthermore, since cryptographic computations often constitute a performance bottleneck in high-speed network implementations, the code for these security-critical components is typically

written in low-level C and assembly and makes use of subtle mathematical optimizations, making it hard to audit and test for developers who are not domain experts.

Formal Verification of Cryptographic Components. In recent years, formal methods for software verification have emerged as effective tools for systematically preventing entire classes of bugs in cryptographic software (see [7] for a survey). For example, verification frameworks like F⋆ [45], EasyCrypt [9], and Coq [10] are used to verify high-performance cryptographic code written in C [48], [29] and assembly [19], [2]. Cryptographic analysis tools like ProVerif [16] and CryptoVerif [17] are used to verify protocols for security properties against sophisticated attackers [14], [12]. Languages like F⋆ have been used to write verified parsers [44] and protocol code [24], [41].

By focusing on specific security-critical components, these tools have been able to make practical contributions towards increasing the assurance of widely-used cryptographic software. For example, verified C code from the HACL⋆ cryptographic library [48] is currently deployed within the Firefox web browser and Linux kernel.

Conversely, the use of domain-specific tools also has its drawbacks. Each verification tool has its own formal specification language tailored for a particular class of analysis techniques. This fragmentation means that we cannot easily compose a component verified against an F⋆ specification with another verified using EasyCrypt. More worryingly, these formal languages are unfamiliar to developers, which can lead to misunderstandings of the verified guarantees and the unintended misuse of the components. For example, an application developer who incorrectly assumes that an elliptic curve implementation validates the peer's public key may decide to skip this check and become vulnerable to an attack [22].

Safely Integrating Verified Code. Without sufficient care, the interface between unverified application code and verified components can become a point of vulnerability. Applications may misuse verified components or compose them incorrectly. Even worse, a memory safety bug or a side-channel leak in unverified application code may reveal cryptographic secrets to the adversary, even if all the cryptographic code is formally verified. A classic example is HeartBleed [27], a memory safety bug in the OpenSSL implementation of an obscure protocol feature, which allowed remote attackers to learn the private keys for thousands of web servers. A similar bug anywhere in the millions of lines of C++ code in the Firefox web browser would invalidate all the verification guarantees of the embedded HACL⋆ crypto code.

To address these concerns, we propose a hybrid framework (depicted in Fig. 1) that allows programmers to safely integrate application code written in a strongly typed programming language with clearly specified security-critical components that are verified using domain-specific tools. We choose Rust as the application programming, and rely on the Rust type system to guarantee memory safety. In addition, we provide a library of secret integers that enforces a *secret independent* coding discipline to eliminate source-level timing side-channels. Hence, well-typed code that only uses secret integers does not break the security guarantees of embedded crypto components.

hacspec: Cryptographic Specifications In Rust. At the heart of our framework is hacspec, a specification language for cryptographic components with several notable features: (1) Syntactically, hacspec is a subset of Rust, and hence is familiar to developers, who can use the standard Rust development toolchain to read and write specifications of cryptographic algorithms and constructions. (2) Specs written in hacspec are executable and so they can be tested for correctness and interoperability, and they can be used as prototype implementations of crypto when testing the rest of the application. (3) The hacspec library includes high-level abstractions for commonly used mathematical constructs like prime fields, modular arithmetic, and arrays, allowing the developer to write succint specifications that correspond closely with the pseudocode descriptions of cryptographic algorithms in published standards. (4) hacspec is a purely functional language without side-effects, equipped with a clean formal semantics that makes specifications easy to reason about and easy to translate to other formal languages like F⋆.

These features set hacspec apart from other crypto-oriented specification languages. The closest prior work that inspired the design of hacspec is hacspec-python [13], a cryptographic specification language embedded in Python that could also be compiled to languages like F⋆. However, unlike Rust, Python is typically not used to build cryptographic software, so specs written in hacspec-python stand apart from the normal developer workflow and serve more as documentation than as useful software components. Furthermore, since Python is untyped, hacspec-python relies on a custom type-checker, but building and maintaining a Python type-checker that provides intuitive error messages is a challenging engineering task. Because of these usability challenges, hacspec-python has fallen into disuse, but we believe that our approach of integrating hacspec specifications into the normal Rust development workflow offers greater concrete benefits to application developers, which increases its chances of adoption.

Contributions. We propose a new framework for safely integrating verified cryptographic components into Rust applications. As depicted in Figure 1, the developer starts with a hacspec specification of a crypto component that serves as a prototype implementation for testing. Then, via a series of compilation and verification steps, we obtain a Rust module that meets the hacspec specification and can be used to replace the hacspec module before the application is deployed. Hence, the programmer can incrementally swap in verified components while retaining full control over the specifications of each component.

Our main contribution is hacspec, a new specification language for security-critical components that seeks to be accessible to Rust programmers, cryptographers, and verification experts. We present the formal syntax and semantics of hacspec, which is the first formalization of a purely functional subset of Rust, to our knowledge. We demonstrate the use of hacspec on a series of popular cryptographic algorithms, but we believe that hacspec can be used more generally to write functional specifications for Rust code. Our second contribution is a set of tools for hacspec, including a compiler from hacspec to F⋆ that enable the safe integration of verified C and assembly code HACL⋆ in Rust applications. Our third contribution is a set

of libraries that any Rust application may use independent of hacspec, including a secret integer library that enforces a constant-time coding discipline, and Rust bindings for the full EverCrypt cryptographic provider [42].

The main technical limitation of our work is that not all the steps are formally verified. To use verified C code from HACL⋆ for example, the programmer needs to trust the compiler from hacspec to F⋆, the Rust binding code that calls the C code from Rust, and the Rust and C compilers. We carefully document each of these steps and intend to formalize and verify some of these elements in the future. In this work, however, we focus on building a pragmatic toolchain that solves a pressing software engineering problem in real-world cryptographic software.

Outline. Section §2 starts by presenting the hacspec language and its implementation. Then, we introduce our security-oriented Rust libraries in section §3, that can be used in conjunction with the hacspec language or in a standalone context. Section §4 deals with the connection between the Rust-based hacspec tooling and multiple verification frameworks having a track record in cryptographic proofs. Finally, we discuss related and future work in section §5. Appendix §A contains the full presentation of hacspec's typing judgment.

2. The hacspec Language

hacspec is a domain-specific language embedded in the Rust programming language and targeted towards cryptographic specifications. It serves several purposes. Firstly, it acts as a frontend for verification toolchains. We provide a formal description of the syntax, semantics and type system of hacspec as a reference, and describe translations from hacspec to specifications in F⋆. Secondly, hacspec aims to be a shared language that can foster communication between cryptographers, Rust programmers and proof engineers.

As a motivating example, consider the ChaCha20 encryption algorithm standardized in RFC 8439 [39]. The RFC includes pseudocode for the ChaCha20 block function in 20 lines of informal syntax (see Fig. 3). However, this pseudocode is not executable and hence cannot be tested for bugs. Indeed, an earlier version of this RFC has several errors in pseudocode. Furthermore, pseudocode lacks a formal semantics and cannot be used as a formal specification for software verification tools.

Fig. 2 shows code for the same function written in hacspec. It has 23 lines of code, and matches the RFC pseudocode almost line-by-line. The code is concise and high-level, but at the same time is well-typed Rust code that can be executed and debugged with standard programming tools. Finally, this code has a well-defined formal semantics and can be seen as a target for formal verification.

We believe that hacspec programs straddle the fine line between pseudocode, formal specification, and prototype implementation and are useful both as documentation and as software artifacts. In this section, we detail the syntax, semantics and type system of hacspec, show how we embed it in Rust, and describe our main design decisions.

```
fn inner_block(state: State) -> State {
    let state = quarter_round(0, 4, 8, 12, state);
    let state = quarter_round(1, 5, 9, 13, state);
    let state = quarter_round(2, 6, 10, 14, state);
    let state = quarter_round(3, 7, 11, 15, state);
    let state = quarter_round(0, 5, 10, 15, state);
    let state = quarter_round(1, 6, 11, 12, state);
    let state = quarter_round(2, 7, 8, 13, state);
    quarter_round(3, 4, 9, 14, state)
}

fn block(key: Key, ctr: U32, iv: IV) -> StateBytes {
    let mut state = State::from_seq(&constants_init()
        .concat(&key_to_u32s(key))
        .concat(&ctr_to_seq(ctr))
        .concat(&iv_to_u32s(iv)));
    let mut working_state = state;
    for _i in 0..10 {
        working_state = chacha_double_round(state);
    }
    state = state + working_state;
    state_to_bytes(state)
}
```

Figure 2. hacspec's version of Chacha20 Block

```
inner_block (state):
    Qround(state, 0, 4, 8,12)
    Qround(state, 1, 5, 9,13)
    Qround(state, 2, 6,10,14)
    Qround(state, 3, 7,11,15)
    Qround(state, 0, 5,10,15)
    Qround(state, 1, 6,11,12)
    Qround(state, 2, 7, 8,13)
    Qround(state, 3, 4, 9,14)
    end

chacha20_block(key, counter, nonce):
    state = constants | key | counter | nonce
    working_state = state
    for i=1 upto 10
        inner_block(working_state)
        end
    state += working_state
    return serialize(state)
    end
```

Figure 3. The ChaCha20 Block Function in Pseudocode (RFC7532, 2.3.1)

2.1. An Embedded Domain Specific Language

hacspec is a typed subset of Rust, and hence all hacspec programs are valid Rust programs. However, the expressiveness of hacspec is deliberately limited, compared to the full Rust language. We believe that a side-effect-free purely-functional style is best suited for concise and understandable specifications. Extensive use of mutable state, as in C and Rust programs, obscures the flow of data and forces programmers to think about memory allocation and state invariants, which detracts from the goal of writing "obviously correct" specifications. Hence, we restrict hacspec to forbid mutable borrows, and we limit immutable borrows to function arguments.

The usual way of writing side-effect-free code Rust is to use .clone() to duplicate values. Figuring out where to insert .clone() calls is notably difficult for new Rust users – in spite of good quality Rust compiler error messages. We intentionally strived to reduce the need for .clone() calls in hacspec. We leverage the Copy trait for all the values

that can be represented by an array of machine integers whose length is known at compile-time. This holds true for all kinds of machine integers, but also for the value types defined in the libraries (later discussed in §3).

hacspec benefits from the strong type system of Rust both to avoid specification bugs and to cleanly separate logically different values using types. For instance, separately declared array types like Key and StateBytes are disjoint, which forces the user to explicitly cast between them and avoids, for instance, the inadvertent mixing of cryptographic keys with internal state.

2.2. Syntax, Semantics, Type System

We describe in detail a simplified version of hacspec. The main simplification lies in the values of the language. We present our formalization with a dummy integer type, but the full hacspec language features all kinds of machine integers, as well as modular natural integers. The manipulation of these other values, detailed in §3, does not involve new syntax or unusual semantics rules, so we omit them from our presentation. Essentially, the rules for binary and unary operators over each kind of machine and natural integers are similar to the rules for our dummy integers; all other operators are modeled as functions and therefore governed by the standard rules about functions.

Syntax. The syntax of hacspec (Fig. 4) is a strict subset of Rust's surface syntax, with all the standard control flow operators, values, and functions. However, hacspec source files are also expected to import a standard library that defines several macros like `array!`. These macros add abstractions like arrays an natural integers to hacspec. The other notable syntactic feature of hacspec is the restriction on borrowing: hacspec only allows immutable borrowings in function arguments. This is the key mechanism by which we are able to greatly simplify the general semantics of safe Rust, compared to existing work like Oxide [47].

Semantics. The structured operational semantics for hacspec corresponds to a simple first-order, imperative, call-by-value programming language. To demonstrate the simplicity of these semantics, we will present them in full here.

The first list of Fig. 5 presents the values of the language: booleans, integers, arrays and tuples. The evaluation context is an unordered map from variable identifiers to values. Here are the different evaluation judgments that we will present:

The second list of Fig. 5 shows the evaluation judgments for the various syntactic kinds of hacspec. The big-step evaluation judgment for expressions $p; \Omega \vdash e \Downarrow v$, reads as: "in program p (containing the function bodies to evaluate function calls) and evaluation context Ω, the expression e evaluates to value v". The other evaluation judgments read in a similar way. The last evaluation judgment is the top-level function evaluation, which is meant as the entry point of the evaluation of a hacspec program.

First, let us examine the simplest rules for values and variable evaluation:

p	::=	$[i]^*$	program items
i	::=	`array!` (t, μ, $n \in \mathbb{N}$)	array type declaration
	|	`fn` f ($[d]^+$) -> μ b	function declaration
d	::=	x : τ	function argument
μ	::=	`unit` | `bool` | `int`	base types
	|	`Seq<` μ `>`	sequence
	|	t	type variable
	|	($[\mu]^+$)	tuple
τ	::=	μ	plain type
	|	&μ	immutable reference
b	::=	{ $[s;]^+$ }	block
s	::=	`let` x : τ = e	let binding
	|	x = e	variable reassignment
	|	`if` e `then` b (`else` b)	conditional statements
	|	`for` x `in` e .. e b	for loop (integers only)
	|	x [e] = e	array update
	|	e	return expression
	|	b	statement block
e	::=	() | `true` | `false`	unit and boolean literals
	|	$n \in \mathbb{N}$	integer literal
	|	x	variable
	|	f ($[a]^+$)	function call
	|	$e \odot e$	binary operations
	|	$\oslash$ e	unary operations
	|	($[e]^+$)	tuple constructor
	|	e . ($n \in \mathbb{N}$)	tuple field access
	|	x [e]	array or seq index
a	::=	e	linear argument
	|	&e	call-site borrowing
$\odot$	::=	+ | - | *	
	|	/ | && | ||	
	|	== | != | > | <	
$\oslash$	::=	- | ~	

Figure 4. Syntax of hacspec

Value	v	::=	() | `true` | `false`
		|	$n \in \mathbb{Z}$
		|	[$[v]^*$]
		|	($[v]^*$)
Evaluation context	Ω	::=	$\varnothing$
(unordered map)		|	$x \mapsto v, \Omega$

Expression evaluation: $p; \Omega \vdash e \Downarrow v$
Function argument evaluation: $p; \Omega \vdash a \Downarrow v$
Statement evaluation: $p; \Omega \vdash s \Downarrow v \Rrightarrow \Omega$
Block evaluation: $p; \Omega \vdash b \Downarrow v \Rrightarrow \Omega$
Function evaluation: $p \vdash f$ ($v_1, \ldots, v_n$) $\Downarrow v$

Figure 5. Values and evaluation judgments of hacspec

$$\textsc{EvalUnit}\ \frac{}{p; \Omega \vdash () \Downarrow ()} \qquad \textsc{EvalBool}\ \frac{b \in \{\texttt{true}, \texttt{false}\}}{p; \Omega \vdash b \Downarrow b} \qquad \textsc{EvalInt}\ \frac{n \in \mathbb{Z}}{p; \Omega \vdash n \Downarrow n}$$

$$\textsc{EvalVar}\ \frac{x \mapsto v \in \Omega}{p; \Omega \vdash x \Downarrow v}$$

We can now move to the rules for function calls evaluation. As shown in the syntax, some immutable borrowing is authorized for function calls arguments. This borrowing is basically transparent for our evaluation semantics, as the following rules show:

$$\textsc{EvalFuncArg}\ \frac{p; \Omega \vdash e \Downarrow v}{p; \Omega \vdash e \Downarrow v} \qquad \textsc{EvalBorrowedFuncArg}\ \frac{p; \Omega \vdash e \Downarrow v}{p; \Omega \vdash \&e \Downarrow v}$$

All values inside the evaluation context Ω are assumed to be duplicable at will. Of course, an interpreter following these rules will be considerably slower compared to the original Rust memory sharing discipline, because it will have to copy a lot of values around. But we argue that this simpler evaluation semantics yields the same results as the original Rust, for our very restricted subset.

We can now proceed to the function call evaluation rule, which looks up the program p for the body of the function called.

$$\textsc{EvalFuncCall}\quad \frac{\texttt{fn}\ f\ (\ x_1\ :\ \tau_1, \ldots, x_n\ :\ \tau_n\ )\ \texttt{->}\ \mu\ b \in p \qquad \forall i \in [\![1, n]\!],\ p; \Omega \vdash a_i \Downarrow v_i \qquad p; x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \vdash b \Downarrow v}{p; \Omega \vdash f\ (\ a_1, \ldots, a_n\ ) \Downarrow v}$$

Next, the evaluation rules for binary and unary operators. These rules are completely standard and assume that the operator has been formally defined on the values of hacspec it can operate on. The dummy arithmetic operators that we have defined in our syntax are assumed to operate only on integers and have the usual integer arithmetic behavior.

$$\textsc{EvalBinaryOp}\quad \frac{p; \Omega \vdash e_1 \Downarrow v_1 \qquad p; \Omega \vdash e_2 \Downarrow v_2}{p; \Omega \vdash e_1 \odot e_2 \Downarrow v_1 \odot v_2} \qquad \textsc{EvalUnaryOp}\quad \frac{p; \Omega \vdash e \Downarrow v}{p; \Omega \vdash \oslash e \Downarrow \oslash v}$$

The rules governing tuples are also very standard. Here, we chose to include only tuple access $e.n$ in our semantics but one can derive similar rules for a tuple destructuring of the form $\texttt{let}\ (\ x_1, \ldots, x_n\ )\ :\ \tau = e$ (which can also be viewed as a syntactic sugar for multiple tuple accesses).

$$\textsc{EvalTuple}\quad \frac{\forall i \in [\![1, n]\!],\ p; \Omega \vdash e_i \Downarrow v_i}{p; \Omega \vdash (\ e_1, \ldots, e_n\ ) \Downarrow (\ v_1, \ldots, v_n\ )}$$

$$\textsc{EvalTupleAccess}\quad \frac{p; \Omega \vdash e \Downarrow (\ v_1, \ldots, v_m\ ) \qquad n \in [\![1, m]\!]}{p; \Omega \vdash e.n \Downarrow v_n}$$

Array accesses are handled similarly to tuple accesses. Note that while the Rust syntax for array access is only a syntactic sugar that calls the .index() function of the Index trait, we view it as a primitive of the language. By giving a dedicated evaluation rule to this construct, we are able to hide the immutable borrowing performed by the .index() function.

$$\textsc{EvalArrayAccess}\quad \frac{p; \Omega \vdash x \Downarrow [\ v_0, \ldots, v_m\ ] \qquad p; \Omega \vdash e \Downarrow n \qquad n \in [\![0, m]\!]}{p; \Omega \vdash x[\ e\ ] \Downarrow v_n}$$

We have completely described the evaluation of expressions, let us now move to statements. The next two rules are similar but correspond to two very different variable assignments. The first rule is a traditional, expression-based let binding that creates a new scope for the variable x. The second rule deals with variable reassignment: x has to be created first with a let-binding before reassigning it. We omit here the difference that Rust does between immutable and mutable variables (mut). Rust has an immutable-by-default policy that helps programmers better spot where they incorporate mutable state, but here we just assume that all variables are mutable for the sake of simplicity. Both statements have a unit return type, to match Rust's behavior.

$$\textsc{EvalLet}\quad \frac{x \notin \Omega \qquad p; \Omega \vdash e \Downarrow v}{p; \Omega \vdash \texttt{let}\ x\ :\ \tau = e \Downarrow ()\ \Rrightarrow x \mapsto v, \Omega}$$

$$\textsc{EvalReassign}\quad \frac{p; x \mapsto v, \Omega \vdash e \Downarrow v'}{p; x \mapsto v, \Omega \vdash x = e \Downarrow ()\ \Rrightarrow x \mapsto v', \Omega}$$

The rules for conditional statements are standard. We do not currently include inline conditional expressions in our syntax, although they are legal in Rust and compatible with hacspec. This restricts the return type of conditional blocks to unit.

$$\textsc{EvalIfThenTrue}\quad \frac{p; \Omega \vdash e_1 \Downarrow \texttt{true} \qquad p; \Omega \vdash b \Downarrow ()\ \Rrightarrow \Omega'}{p; \Omega \vdash \texttt{if}\ e_1\ b \Downarrow ()\ \Rrightarrow \Omega'}$$

$$\textsc{EvalIfThenFalse}\quad \frac{p; \Omega \vdash e_1 \Downarrow \texttt{false}}{p; \Omega \vdash \texttt{if}\ e_1\ b \Downarrow ()\ \Rrightarrow \Omega}$$

$$\textsc{EvalIfThenElseTrue}\quad \frac{p; \Omega \vdash e_1 \Downarrow \texttt{true} \qquad p; \Omega \vdash b \Downarrow ()\ \Rrightarrow \Omega'}{p; \Omega \vdash \texttt{if}\ e_1\ b\ \texttt{else}\ b' \Downarrow ()\ \Rrightarrow \Omega'}$$

$$\textsc{EvalIfThenElseFalse}\quad \frac{p; \Omega \vdash e_1 \Downarrow \texttt{false} \qquad p; \Omega \vdash b' \Downarrow ()\ \Rrightarrow \Omega'}{p; \Omega \vdash \texttt{if}\ e_1\ b\ \texttt{else}\ b' \Downarrow ()\ \Rrightarrow \Omega'}$$

Looping is very restricted in hacspec, since we only allow for loops ranging over an integer index. This restriction is purposeful, since general while loops can be difficult to reason about in proof assistants. One could also easily add a construct looping over each element of an array, which Rust already supports. However, we chose not to include the idiomatic .iter().map() calls to avoid closures.

$$\textsc{EvalForLoop}\quad \frac{p; \Omega \vdash e_1 \Downarrow n \qquad p; \Omega \vdash e_2 \Downarrow m \qquad \Omega_n = \Omega \qquad \forall i \in [\![n, m-1]\!],\ p; x \mapsto i, \Omega_i \vdash b \Downarrow ()\ \Rrightarrow \Omega_{i+1}}{p; \Omega \vdash \texttt{for}\ x\ \texttt{in}\ e_1\ \texttt{..}\ e_2\ b \Downarrow ()\ \Rrightarrow \Omega_m}$$

The array update statement semantics are standard. Here, we require that e_1 evaluates to an in-bounds index. We chose to omit the error case where the index falls outside the range of the array, to avoid including a classic propagating error in our semantics. In Rust, the default behavior of out-of-bounds indexing is to raise a panic!() that cannot be caught. Like array indexing, array updating is treated by Rust as a syntactic sugar to a .index_mut() call, which mutably borrows its argument. We treat this syntactic sugar as a first-class syntactic construct in hacspec to carefully specify the behavior of the underlying mutable borrow of the array.

$$\textsc{EvalArrayUpd}\quad \frac{p; x \mapsto [\ v_0, \ldots, v_n\ ], \Omega \vdash e_1 \Downarrow m \qquad m \in [\![0, n]\!] \qquad p; x \mapsto [\ v_0, \ldots, v_n\ ], \Omega \vdash e_2 \Downarrow v}{p; x \mapsto [\ v_0, \ldots, v_n\ ], \Omega \vdash x[\ e_1\ ] = e_2 \Downarrow ()\ \Rrightarrow x \mapsto [\ v_0, \ldots, v_{m-1}, v, v_{m+1}, \ldots, v_n\ ], \Omega}$$

The next two rules replicate the special case of the last statement of a block in Rust. Indeed, the return keyword in Rust is optional for the last statement of a function, because the value returned by a function is assumed to be the result of the expression contained in the last statement.

In fact, our syntax does not include the return keyword at all to avoid control-flow-breaking effects.

$$\textsc{EvalExprStmt}\ \frac{p; \Omega \vdash e \Downarrow v}{p; \Omega \vdash e \Downarrow v \Rrightarrow \Omega}$$

Blocks in Rust are a list of statements, which also act as a scoping unit: a variable defined inside a block cannot escape it. This behavior is captured by the intersection of the two contexts at then end of EvalBlockAsStatement: we keep all the values Ω' that were already defined in Ω.

$$\textsc{EvalBlock}\ \frac{p; \Omega \vdash s_1 \Downarrow () \Rrightarrow \Omega' \quad p; \Omega' \vdash \{ s_2; \ldots; s_n \} \Downarrow v \Rrightarrow \Omega''}{p; \Omega \vdash \{ s_1; \ldots; s_n \} \Downarrow v \Rrightarrow \Omega''}$$

$$\textsc{EvalBlockOne}\ \frac{p; \Omega \vdash s \Downarrow v \Rrightarrow \Omega'}{p; \Omega \vdash \{ s \} \Downarrow v \Rrightarrow \Omega'} \qquad \textsc{EvalBlockAsStatement}\ \frac{p; \Omega \vdash b \Downarrow v \Rrightarrow \Omega'}{p; \Omega \vdash b \Downarrow v \Rrightarrow \Omega' \cap \Omega}$$

Finally, we can define the top-level rule that specifies the execution of a function, given the values of its arguments.

$$\textsc{EvalFunc}\ \frac{\texttt{fn}\ f\ (\ x_1 : \tau_1, \ldots, x_n : \tau_n\ )\ \texttt{->}\ \mu\ b \in p \quad p; x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \vdash b \Downarrow v}{p \vdash f\ (\ v_1, \ldots, v_n\ ) \Downarrow v}$$

Typing. While the operational semantics of hacspec are simple, its typing judgment is trickier. This judgment has to replicate Rust's typechecking on our restricted subset, including borrow-checking. The other complicated part of Rust typechecking, trait resolution, is currently out of hacspec's scope, even though some polymorphism could be introduced in future work to hacspec via a limited use of Rust traits.

The typing judgment for expressions is the following:

$$\Gamma; \Delta \vdash e : \tau \Rrightarrow \Gamma'$$

Δ is a dictionary mapping type names to their definition, since Rust's type system is nominal rather than structural. Γ is the important context, since it maps variables to their types. The typing judgment saying that e has type τ also returns a new context Γ' because of linearity. Rust's type system contains some linearity as the ownership of a non-borrowed variable is exclusive. Using a linear variable is akin to a "move" in Rust's terminology. To replicate this linearity and match Rust's typechecking behavior, hacspec's typechecking context Γ is modified by the typechecking as some variables may be consumed: $\Gamma' \subset \Gamma$.

The full typechecking semantics can be found in §A. It largely follows the traditional structure of a linear typechecking judgment. The tricky part concerns the handling of immutable borrowing for function arguments. Here are the rules:

$$\textsc{TypFuncCall}\ \frac{f : (\ \mu_1, \ldots, \mu_n\ ) \to \tau \in \Gamma \qquad \Delta \vdash \Gamma = \Gamma_1 \circ \cdots \circ \Gamma_n \quad \forall i \in [\![1, n]\!],\ \Gamma_i; \Delta \vdash a_i \sim \mu_i \Rrightarrow \Gamma_i' \quad \Delta \vdash \Gamma' = \Gamma_1' \circ \cdots \circ \Gamma_n'}{\Gamma; \Delta \vdash f\ (\ a_1, \ldots, a_n\ ) : \tau \Rrightarrow \Gamma'}$$

```
foo(&(x,x))
      - ^ value used here after move
      |
      value moved here
```

Figure 6. Rust error message for function argument borrowing

$$\textsc{TypFunArg}\ \frac{\Gamma; \Delta \vdash e : \tau \Rrightarrow \Gamma'}{\Gamma; \Delta \vdash e \sim \tau \Rrightarrow \Gamma'} \qquad \textsc{TypFunArgBorrow}\ \frac{\Gamma; \Delta \vdash e : \mu \Rrightarrow \Gamma'}{\Gamma; \Delta \vdash \&e \sim \&\mu \Rrightarrow \Gamma'}$$

$$\textsc{TypFunArgBorrowVar}\ \frac{\Gamma; \Delta \vdash x : \mu \Rrightarrow \_}{\Gamma; \Delta \vdash \&x \sim \&\mu \Rrightarrow \Gamma}$$

$\Delta \vdash \Gamma = \Gamma_1 \circ \cdots \circ \Gamma_n$ corresponds to a linear split of the context. TypFuncCall follows a standard pattern for linear typing: it splits the context between the arguments of the function, typechecks the arguments and builds a new context out of the argument's resulting context.

While TypFunArg and TypFunArgBorrow are transparent, TypFunArgBorrowVar is the trickiest rule to explain. Let us imagine you are calling the function foo(&(x,x)) where x is a non-borrowed, non-copyable value. The Rust compiler will give you the error message of Fig. 6.

This means that even though the argument of the function is borrowed, the borrowing happens after the typechecking of the borrowed term using regular rules and linearity. However, if you were to typecheck foo(&x, &x) Rust's typechecker would not complain because the borrowing directly affects x, and not an object of which x is a part of.

This unintuitive feature of the type system becomes more regular when looking at a desugared representation of the code like MIR. But in our type system, we have to include several rules like TypFunArgBorrowVar to reflect it. Since we deal with the surface syntax of Rust, these special rules for borrows are necessary. Although hacspec does not currently include structs, we expect special rules to be added to deal with borrowing struct fields. Note that the syntax of hacspec only allows array indexing to be done via $x\ [\ e\ ]$ instead of the more general $e_1\ [\ e_2\ ]$, with the objective of keeping rules simple, since indexing implies borrowing in Rust.

2.3. Implementation

We implement the syntax and type system presented above in the hacspec typechecker. Programmers need tooling that help them stay within the bounds of the language, and it is precisely the role of the hacspec typechecker, which kicks in after the regular Rust typechecker.

Compiler Architecture. The hacspec typechecker is completely implemented in Rust, integrated into the regular Rust ecosystem of cargo and the Rust compiler. As such, programmers that already use Rust need not to install complex dependencies to use hacspec. Concretely, the hacspec typechecker uses the rustc_driver [1] crate, offering direct access to the Rust compiler API. The Rust

1. https://doc.rust-lang.org/nightly/nightly-rustc/rustc_driver/

```
error[hacspec]: type not allowed in hacspec
  --> hacspec-poly1305/src/poly1305.rs:61:17
   |
61 | pub fn poly(m: &[U8], key: KeyPoly) -> Tag {
   |                 ^^^^
```

Figure 7. hacspec error message

compiler is architected as a series of translations between several intermediate representations:

$$\text{AST} \xrightarrow{\text{desugaring}} \text{HIR} \xrightarrow{\text{to CFG}} \text{MIR} \xrightarrow{\text{to LLVM}} \text{LLVM IR}$$

The borrow checking and typechecking are bound to be performed exclusively on MIR, making it the richest and most interesting IR to target as a verification toolchain input. However, MIR's structure is quite far from the original Rust AST. For instance, MIR is control-flow-graph-based (CFG) and its control flow is destructured, making it hard for a deductive verification toolchain to recover the structure needed to host loop invariants. For hacspec, we chose to take the Rust AST as our input, for two reasons.

First, choosing the AST moves the formalization of hacspec closer to the Rust source code, making it easier to understand. Second, it enables a compilation to the target verification toolchains that preserves the overall look of the code, easing the transition from Rust to the verified backends, and the communication between a cryptography expert and a proof engineer. This choice makes hacspec quite different from the other Rust-based verification tools discussed in §5.2.

Three-tier Typechecking. The hacspec typechecker operates in three phases.

In the first phase, the program goes through its regular flow inside the Rust compiler, up to regular Rust typechecking. Second, the Rust surface AST is translated into a smaller AST, matching the formal syntax of hacspec. Third, a typechecking phase following the formal typechecking rules of hacspec is run on this restricted AST.

The second phase is the most helpful for the developers, as it will yield useful error messages whenever the program does not fall withing the hacspec subset of Rust. The error messages look like regular error messages emitted by the Rust compiler, as shown in figure Fig. 7.

Here, the error message points at the native Rust slice being used, which is forbidden since native Rust slices are not part of hacspec (the corresponding type is ByteSeq). These error messages are a key component of the security-oriented aspect of hacspec, since they enforce the subset limits at compile-time rather than at execution time. The goal of this tooling is to make it impossible for the programmer to unknowingly break the abstraction of the subset.

The third phase, corresponding to hacspec's typechecking, should never yield any error since the program has at this point already be type-checked by Rust. If the program had a typing problem, it should have been caught by the Rust typechecker first and never come to this third phase. However, the hacspec typechecker still catches some restrictions of hacspec subset that are not purely syntactic, and therefore pass the second phase.

Interacting With The Rust typechecker. The first kind of errors caught by the third phase concerns external function calls. Indeed, one should only call in hacspec functions that are within the bounds of hacspec. But this rule suffer some exceptions, the main one being the primitive functions of the hacspec library that operate on base types such as Seq, whose behavior is part of the trusted computing base. Another issue is the import of functions defined outside of the current crate. The import of the hacspec library is recognized by the hacspec typechecker and given special treatment, but we also allow importing regular crates. This allows hacspec programs to enjoy some kind of modularity, as programs can be distributed over several crates. The hacspec typechecker uses the Rust compiler's "crate metadata reader" to import function definitions from external crates, and scan their signatures. If the signature of an imported function typechecks in hacspec, then its use is valid and does not yield an error. This behavior, while practical, leaves a gap opened for breaking the abstraction of the subset. Indeed, one could import a function whose signature is in hacspec, but its body is not, thereby increasing the trusted computing base. A solution to this problem would be to define an allow-list of valid hacspec functions via Rust's attribute system, but technical constraints on the Rust compiler (custom attributes are erased from crate metadata) makes this solution inoperable. A better system to close this potential abstraction leak is left as future work.

The second kind of errors yielded by the third phase of hacspec program processing relates to type inference. The hacspec typechecking procedure does not support any kind of type inference, unlike regular Rust typechecking. Nevertheless, idiomatic Rust programs often rely on type inference for things like integer literal types, or methods operating on parametrized types. Hence, hacspec forces programmers to explicitly annotate their integer literals with their type, using regular Rust syntax like 1u32. This issue also arises with methods. hacspec's syntax does not include methods because a method call is the same as calling a function whose first argument is the self. However, this assumes that method resolution has already been performed. This is not the case when taking the Rust AST as an input, which means that the hacspec typechecker has to replicate the Rust method resolution algorithm. The algorithm is very simple: it typechecks the first self argument, then looks into a map from types to functions if the type contains the correct function being called. This behavior is more complicated with parametric types such as Seq, introducing a nested map for the type parameter, but keeps the same principle. Hence, and because of the lack of type inference, the programmer has to explicitly annotate the type parameter of methods concerning parametric types, using regular Rust syntax like Seq::<U8>::new(16).

3. The hacspec Libraries

The hacspec language alone is not sufficient to write meaningful programs. Indeed, cryptographic and security-oriented programs manipulate a wide range of values that need to be represented in hacspec. To that purpose, we provide several Rust standalone libraries that implement security-related abstractions. Although these libraries are

tightly integrated in hacspec, they can be used individually in regular Rust applications.

3.1. Secret Integers

Timing side-channels are some of the most important threats to cryptographic software, with many CVEs for popular libraries appearing every year. To mitigate these attacks, cryptographic code is usually written to be *secret independent*: the code never branches on a secret value and it never uses a secret index to access an array. This discipline is sometimes called *constant time coding* and many verification tools try to enforce these rules at compile time or in the generated assembly [8], [3].

We follow the approach of HACL⋆ [48] to enforce secret indendence at the source-code level using the standard Rust typechecker. We build a library of secret machine integers that contains wrapped versions of all machine integer types, signed and unsigned, supported by Rust. These wrappers define a new type U8, U32,... corresponding to each integer type u8, u32,... in Rust. To define our secret integer types, we make use of Rust's struct declaration semantics and nominal typing, which differs from a simple type alias introduced with type.

For each integer type, the library defines only the operations that are known to be constant-time, *i.e.* operations that are likely to be compiled to constant-time instructions in most assembly architectures. Hence, secret integers support addition, subtraction, multiplication (but not division), shifting and rotation (for public shift values), bitwise operators, conversions to and from big- and little-endian bytearrays, and constant-time masked comparison.

Secret integers can be converted to and from regular Rust integers, but only by calling an explicit operation. The library provides the functions classify and declassify, to make visible in the code the points where there is an information flow between public and secret. Hence, a Rust program that uses secret integers but never calls declassify is guaranteed to be secret independent at the source-code level. We have carefully curated the integer operations to ensure so that the Rust compiler should preserve this guarantee down to assembly, but this is not a formal guarantee, and the generated assembly should still be verified using other tools. However, our methodology provides early feedback to the developer and allows them to eliminate an entire class of sofware side-channels.

Secret integers are extensively used in hacspec programs to check that our code does not inadvertently leak secrets. However, these libraries can also be used outside hacspec to add protections to any Rust program. The developer simply replaces the types all potentially secret values (keys, messages, internal cryptographic state) from u8,u32,... to U8,U32,... and uses the Rust typechecker to point out typing failures that may indicate timing leaks. The developer can then selectively add declassify in cases that they carefully audit and deem safe. To the best of our knowledge, this is the first static analysis technique that can analyze Rust cryptographic code for secret independence.

3.2. Fixed-length Arrays

Formal specifications of software components often need to use arrays and sequences, but for clarity of specification and ease of translation to purely functional specifications in languages like Coq and F⋆, we believe that hacspec programs should only use arrays in simple ways. Any sophisticated use of mutable or extensible arrays can quickly make a specification muddled and bug-prone and its invariants harder to understand. Consequently, in hacspec, we only allow fixed-length arrays that cannot be extended (unlike the variable-size Vec type provided by Rust).

The hacspec library includes two kinds of fixed-length arrays. The go-to type is Seq, which models a fixed-length sequence of any type. Seq supports a large number of operations such as indexing, slicing and chunking. The Rust typechecker ensures that the contents of the Seq have the correct type, and C-like array pointer casting is forbidden. This forces the user to properly cast the contents of the array rather than casting the array pointer itself, which can be a source of bugs.

However, Seq has a blind spot triggered by a lot of usual cryptographic specifications bugs: array bounds checking. Seq, like Vec, will trigger a dynamic error if one tries to access an index outside its bounds. This Rust dynamic error is better than the C undefined behavior (usually resulting in a segfault), but is not enough for our security-oriented goals. A full proof that array accesses are always within bounds typically requires the use of a fully-fledged proof assistant, as we'll see in §4. But the hacspec libraries offers a mechanism to bake into Rust's typechecking part of this array bound proof.

This mechanism is offered by the `array!` macro. array!(State, U32, 16) is the declaration of a named Rust type that will correspond to an array of 16 secret 32-bit (secret) integers. The length of 16 is baked into Rust's type-system via the underlying use of the [U32;16] Rust native array type. Using such a native known-length array type compared to a regular Seq has multiple advantages.

First, the array!(State, U32, 16) macro call defines all the methods of the State type on-the-spot, and uses its length to provide debug assertions and dynamic error messages that help the user with the extra information that State's length should always be 16, whereas a Seq can have any length. Moreover, the State constructor expects a literal Rust array of 16 integers, whose length can be directly checked by Rust's typechecker.

Second, the use of array!(State, U32, 16) acts as a length hint to the verification backends of hacspec. Known-length arrays act as control points in the proof that help inference of array lengths during the array bounds proof.

While the two first advantages help increase the correctness of the hacspec program, they often come as a burden to programmers which have to annotate their program with explicit casts between Seq and array!-declared types. However, a third advantage of the underlying native known-length array type compensates the annoyance. Indeed, Seq, as Vec, does not implement the Copy trait and therefore has to be explicitly .clone() every time it is used multiple times without borrowing. As mutable borrowing is forbidden in hacspec, this would lead to a high-number of .clone() calls for Seq values mutated and used in the programs. But since array! uses Rust's native known-length array that implement Copy, their manipulation does not require any explicit .clone() call. This feature is

```
public_nat_mod!(
    type_name: FieldElement,
    type_of_canvas: FieldCanvas,
    bit_size_of_field: 131,
    modulo_value:
      "03ffffffffffffffffffffffffffffffb"
);
```

Figure 8. Declaration of a (public) modular natural number in hacspec

especially helpful for cryptographic code, which usually manipulates small-sized chunks (represented using array!) coming from a few big messages (represented using Seq).

3.3. Modular Natural Integers

Many cryptographic algorithms rely on mathematical constructs like finite fields, groups, rings, and elliptic curves. Our goal is to provide libraries for all these constructions in hacspec, so that the programmer can specify cryptographic components using high-level concepts, without worrying about how to implement them.

For example, one of the most common needs for cryptographic code is modular arithmetic, that is the field of natural numbers between 0 and n with all arithmetic operations performed modulo n. For example, by setting n to a power of 2, we can build large integer types like u256; by setting it to a prime, we obtain a prime field; by choosing a product of primes, we get an RSA field etc.

We provide a dedicated library for arbitrary-precision modular natural integers, that can be manipulated by Rust programs just like machine integer values, without worrying about any allocation or deallocation. Fig. 8 shows what a finite field declaration looks like, taken from the Poly1305 specification. The public_nat_mod! macro call defines a fresh Rust type, FieldElement, along with multiple methods corresponding to operations on these natural integers. The two next arguments of the macro call concern the underlying representation of the natural integer.

Our implementation of modular arithmetic relies on Rust's BigInt crate[2]. But BigInt does not implement the Copy trait and is therefore cumbersome to use, requiring the insertion of numerous .clone() calls. To bypass this limitation, hacspec's modular integers use a concrete representation as a big-endian, fixed-length array of bytes. The length of this array is determined using the bit_size_of_field argument of the macro call. The methods of FieldElement constantly switch back and forth between the Copyable array representation and its BigInt equivalent to get the computations right.

The modulo_value argument contains the value for the modulus n as a hex string because the value can be arbitrarily large and often cannot fit inside a u128 literal. The type_of_canvas: FieldCanvas argument is required merely because of Rust's macro system fundamental limitation of forcing the user to explicitly provide the identifier for all the types declared by the macro. Indeed, the macro defines two types: FieldCanvas is the type for the underlying array-of-bytes representation of the bounded natural integers, that enjoys the Copy trait. FieldElement is a wrapper around FieldCanvas that takes the modulo value into account for all its operations.

hacspec's natural modular integers also come in two versions, public and secret. The secret version can only be converted to arrays of secret integers, ensuring the continuity of information flow checking across machine and natural integers. However, the underlying modular arithmetic arithmetic operations themselves are not constant-time, so this inter-conversion serves primarily to document information flow, not to enforce secret independence.

The seamless interoperability provided by hacspec between machine integers and modular natural integers allows programmers to mix in different styles of specifications, ranging from high-level math-like to low-level implementation-like code. hacspec allows programmers to write and test code at both levels and bridge the gap between them, by allowing them to interoperate, and also though formal verification, as we'll see in §4.

4. Connecting hacspec To Theorem Provers

hacspec is a security-oriented domain-specific language embedded in Rust, along with a set of useful libraries for cryptographic specifications. However, as strong as Rust type system is, it is not sufficient to catch common errors like array indexing problems at compile time. To increase the level of assurance on the correctness of specifications written in hacspec, we implement one backends from hacspec to state-of-the art verification frameworks: F⋆ [45] (§4.1). By using this backend, that could easily be extended to other frameworks like and Easycrypt [9] (§4.4), the hacspec programmer can further debug specifications, find non-trivial errors which may have escaped manual audits, and formally connect the specification to an optimized verified implementation (§4.2). We demonstrate this approach on a library of classic cryptographic primitives (§4.3).

4.1. Translating hacspec to F⋆

The functional semantics of the hacspec language makes it easy to compile it to the F⋆ language. To illustrate this claim, Fig. 9 and Fig. 10 show what the same function, the main loop of the Poly1305 specification, looks like both in hacspec and after its translation to F⋆.

The translation is very regular, as each hacspec assignment is translated to an F⋆ let-binding. Variable names are suffixed with indexes coming from a name resolution pass happening in the hacspec typechecker, to ensure that the scoping semantics are preserved by the compilation. Apart from syntax changes, the bulk of this translation relates to the functional purification of the mutable variables involved in loop and conditional statements. This approach is similar to the earlier work of Electrolysis [46], although much simpler because we do not deal with mutable references.

However, hacspec features mutable plain variables which can be reassigned throughout the program. These reassignments are always translated into functional let-bindings, but since statements are translated into expressions during the compilation to F⋆, the assignment side-effects have to be hoisted upwards in the let-binding corresponding to conditional or loop statements. For example,

2. https://crates.io/crates/bigint

```
1  pub fn poly(m: &ByteSeq, key: KeyPoly) -> Tag {
2    let r = le_bytes_to_num(&key.slice(0, BLOCKSIZE));
3    let r = clamp(r);
4    let s = le_bytes_to_num(
5      &key.slice(BLOCKSIZE, BLOCKSIZE));
6    let s = FieldElement::from_secret_literal(s);
7    let mut a = FieldElement::from_literal(0u128);
8    for i in 0..m.num_chunks(BLOCKSIZE) {
9      let (len, block) =
10       m.get_chunk(BLOCKSIZE, i);
11     let block_uint = le_bytes_to_num(&block);
12     let n = encode(block_uint, len);
13     a = a + n;
14     a = r * a;
15   }
16   poly_finish(a, s)
17 }
```

Figure 9. Poly1305 main loop in hacspec

```
1  let poly (m_15: byte_seq) (key_16: key_poly) : tag =
2    let r_17 = le_bytes_to_num
3      (array_slice (key_16) (usize 0) (blocksize))
4    in
5    let r_18 = clamp (r_17) in
6    let s_19 = le_bytes_to_num
7      (array_slice (key_16) (blocksize) (blocksize))
8    in
9    let s_20 = nat_from_secret_literal
10     (0x03ffffffffffffffffffffffffffffffb) (s_19)
11   in
12   let a_21 = nat_from_literal
13     (0x03ffffffffffffffffffffffffffffffb) (pub_u128 0x0)
14   in
15   let (a_21) = foldi
16     (usize 0) (seq_num_chunks (m_15) (blocksize))
17     (λ i_22 (a_21) →
18       let (len_23, block_24) =
19         seq_get_chunk (m_15) (blocksize) (i_22)
20       in
21       let block_uint_25 = le_bytes_to_num (block_24) in
22       let n_26 = encode (block_uint_25) (len_23) in
23       let a_21 = (a_21) +% (n_26) in
24       let a_21 = (r_18) *% (a_21) in
25       (a_21))
26     (a_21)
27   in
28   poly_finish (a_21) (s_20)
```

Figure 10. Poly1305 main loop in F*, compiled from Fig. 9

see the translation of the loop statement lines 8 to 15 of Fig. 9, to the loop expression lines 15 to 27 of Fig. 10.

While conditional and loop statements constitute the main structural changes of the translation, most of the compiler implementation work goes into connecting the libraries handling arithmetic and sequences in F*. Fortunately, this task is simplified by having access to the rich typing environment provided by the hacspec typechecker, which allows us to insert annotations and hints into the generated F*, significantly easing the out-of-the box typechecking of the translated programs. For instance, FieldElement::from secret literal(s) (line 6 of Fig. 9) is translated to the F* expression nat_from_secret_literal (0x03ff[...]ffb) (s_19) (lines 9-10 of Fig. 10). The modulo value of the FieldElement integer type has been automatically added during the translation, as a hint to F* typechecking.

Overall, these annotations added automatically during

```
1  for i=1 upto ceil(msg length in bytes / 16)
2    n = le_bytes_to_num(
3      msg[((i-1)*16)..(i*16)] | [0x01])
4    a += n
5    a = (r * a) % p
6  end
```

```
11 let block_uint = le_bytes_to_num(
12   &m.slice(BLOCKSIZE * i, BLOCKSIZE));
```

```
21 let block_uint_1876 = le_bytes_to_num (
22   seq_slice (m_1868)
23     ((blocksize) * (i_1875))
24     (blocksize))
25 in
```

```
(Error 19) Subtyping check failed;
expected type len: uint_size{
  blocksize * i_22 + len <= Lib.Seq.length m_15
};
got type uint_size; The SMT solver could not prove
the query, try to spell your proof in more detail
or increase fuel/ifuel
```

Figure 11. RFC 8439: first, the main Poly1305 loop pseudocode containing the bug. Then in second and third, corresponding buggy snippets in hacspec and F* (line numbers of Fig. 9 and Fig. 10). Fourth: F* error message catching the bug

the translation enable smooth typechecking and verification inside F*. The only manual proof annotations still needed in F* concern logical pre-conditions and loop invariants that cannot be expressed using the Rust type system.

4.2. The Benefits Of Using Theorem Provers

Finding Specifications Bugs. Once embedded in F*, various properties out of reach of the Rust type system can be proven correct about the specification. The most obvious of these properties is the correctness of array indexing.

RFC 8439 [39] is the second version of the ChaCha20Poly1305 RFC, written after a number of errata were reported on the previous version. However, the second version still contains a specification flaw that illustrates the need for debugging specifications with proof assistants. RFC 8439 defines the core loop in Poly1305 in a way that overruns the message if its length is not a multiple of 16. Fig. 11 shows the RFC8439 pseudocode, as well as its corresponding snippets in hacspec and F*.

In our hacspec code, the get_chunk function always provides the correct chunk length, preventing the bug. However, if we tried to precisely mimic the RFC pseudocode, we would have introduced the bug, which would not have been caught by the Rust or hacspec typecheckers. The issue could have been uncovered by careful testing, but we note that the standard test vectors did not prevent the introduction of this bug in the RFC.

We claim that using F* to typecheck specifications can help optimize the specification development workflow by catching this kind of bugs early. The bug of RFC 8439 is indeed detected by F* with a helpful error message. hacspec offers an integrated experience for cryptographic specification development: a cryptographer can write a

Primitive / Lines of code	hacspec	HACL⋆
ChaCha20	132	191
Poly1305	77	77
Chacha20Poly1305	59	89
NTRU-Prime	95	–
SHA3	173	227
Curve25519	107	124
SHA256	148	219*

Figure 12. Cryptographic Primitives written in hacspec, lines of code count. The HACL⋆ column is included as reference. *: the HACL⋆ SHA2 specification covers all versions of SHA2, not just 256.

Rust-embdded hacspec program that looks like pseudocode, and do a first round of debugging with testing since hacspec programs are executable. Then, the cryptographer can translate the specification to F⋆ where a proof engineer can prove array-indexing and other properties correct.

Proving Functional Equivalence. Once the specification has been fixed and proven correct, an optimized implementation is required for the cryptographic primitive or protocol to be embedded in a real-world application. There is often a wide gap between the specification and the optimized code, and this spot is where formal methods have proven their usefulness in the past. Proof assistants like F⋆ enable proving the functional equivalence between a specification and an optimized implementation. hacspec fits nicely into this process, since it directly provides the specification in the language of the proof assistant (here F⋆), based on hacspec code that can be audited by cryptographers.

4.3. A Library Of Cryptographic Specifications

To evaluate the hacspec language, libraries, typechecker and compiler, we build a library of popular cryptographic algorithms in hacspec, presented in Fig. 12. For each primitive, we wrote a hacspec specification that typechecks in Rust and with the hacspec typechecker. Then, we translated some of these primitives to F⋆ and typechecked the result with the proof assistant.

Some annotations (maximum 2-3 per primitive) are needed to typecheck the specifications in the proof assistants. These annotations concern bounds for array indexes passed as function parameters, as well as some loop invariants. In the future, we may extend the Rust-embedded syntax of hacspec to include design-by-contracts annotations. This functionality is already provided by crates like contracts[3], where the annotations are incorporated into Rust debug_assert! assertions.

These hacspec specifications have been extensively tested and can be used as prototype crypto implementations when building and debuging larger Rust applications. Further, they form the basis for our verification workflow.

4.4. Translating to EasyCrypt and other Tools

While proof assistants are powerful, they are often specialized. In the cryptography space, Fiat-crypto uses Coq and generate C implementations for elliptic curves [28], and Jasmin covers more kinds of primitives using EasyCrypt [9] but only targets assembly [2]. HACL⋆ [48] and Vale [19], both using the F⋆ proof assistant, are currently the only instance of proven-correct interoperability between C and assembly [32].

In this context, hacspec is a way to break the integration silo imposed by proof assistant frameworks which typically cannot interoperate. The simplicity of the hacspec language semantics, close to a first-order functional language, makes it easy to translate in the specification languages of most proof assistants.

To demonstrate this claim, we present a second backend to the hacspec implementation, targeting EasyCrypt. The translation from hacspec to EasyCrypt is very similar to the translation from hacspec to F⋆. Actually, the main difficulty of implementing this new backend lies in connecting the libraries expected by hacspec to the existing libraries of the proof assistant. The resulting EasyCrypt specification can then be related to optimized assembly code written in Jasmin [4]. Indeed, the implementations of ChaCha20 and Poly1305 in Jasmin rely on an EasyCrypt specification that was manually transpiled from the HACL⋆ specifications for these primitives. Our tools make it possible to automate this step by directy translating the hacspec specifications to both F⋆ and EasyCrypt, and safely compose implementations from HACL⋆ and Jasmin.

5. Related And Future Work

5.1. Cryptographic Specification Languages

hacspec is not the first domain-specific language targeting cryptographic specifications. The most notable works in this domain include Cryptol [30], Jasmin [2], and Usuba [37]. There are two main differences between hacspec and these languages.

The first difference is the embedded nature of hacspec's language. By leveraging the existing Rust ecosystem, we believe it is easier for programmers to use our language, compared to the effort of learning a new domain-specific language with its different syntax. We extend this claim to the tooling of the domain-specific language, which is written completely in the host language (Rust) and therefore does not require installing any extra dependency with whom the developer might be unfamiliar (like an entire OCaml or Haskell stack).

The second difference is the target of the domain-specific language. Cryptol targets C and VHDL, Jasmin targets assembly and Usuba targets C. Cryptol and Jasmin each are closely integrated with their respective proof assistants/verification backend: SAW [26] and Easycrypt [9]. The code written in those domain-specific languages is closer to an implementation than a specification, as it is directly compiled to a performant target and is sometimes proven correct against a more high-level specification (like libjc [4]). Instead, hacspec can target different verification toolchains and acts as a bridge between those projects.

5.2. Rust-based Verification Tools

hacspec is not the first attempt at a Rust frontend for verification toolchains; we provide a summary of existing work in Fig. 13. The "input" column of the comparison table refers to the entry point of the verification frameworks withing the Rust compiler architecture, as discussed in §2.3.

3. https://crates.io/crates/contracts

Frontend	Target(s)	Input	Formal
KLEE [35]	KLEE [21]	MIR	○
crux-mir [33]	SAW [26]	MIR	○
Prusti [5]	Viper [38]	MIR	◑
Electrolysis [46]	Lean [23]	MIR	◑
SMACK [6]	SMACK [43]	LLVM IR	◑
RustHorn [36]	CHC [15]	MIR	●
μMIR [25]	WhyML [31]	MIR	●
hacspec	F* [45], Easycrypt [9]	AST	◐

○ = DSL and translation to target not formalized
◑ = DSL defined by its translation to formalized target
◐ = DSL formalized but not the translation to target(s)
● = DSL and translation to target formalized

Figure 13. Rust frontends for verification toolchains

We also classify the previous work according to the extent of formalization that they contain. Indeed, multiple things can be formalized when creating an embedded domain-specific language. First, the domain-specific language can be defined as a subset of a formalization of the host language. In the case of Rust, only RustBelt [34] currently provides a full formalization. But no existing tool that uses RustBelt can extract to another target. On the other hand, the domain-specific language can be defined intrinsically in terms of its encoding in the formalized target (◑). Finally, the domain-specific language itself can be formalized (◐), as well as its translation to the formalized target (●).

We intend for hacspec to belong to the last category, corresponding to the ● case. However, we chose to prioritize the interoperability of hacspec by targeting multiple backends, which increases the workload of translation formalization. Hence, we leave the migration from ◐ to ● as future work.

The main difference of hacspec compared to previous Rust frontend is the scope of the subset it intends to capture. Indeed, hacspec does not deal with mutable state and borrows, which is the heart of Rust. On the contrary, hacspec explicitly forbids these as its aim is to capture the functional, pure subset of Rust. Of course, this makes it unsuitable for verifying any kind of performance-oriented programs. Instead, we believe such programs should be dealt with using the methodology described in §**??**. hacspec provides to Rust programmers an entry point into the verification world, inside the ecosystem that they are familiar with.

5.3. Future work

There are two main directions for future work on hacspec. The first direction concern its usability as a set of tools to write security-oriented Rust specifications. The hacspec language could be augmented with idiomatic Rust features that do not break its functional core and semantics, like algebraic data types (structs and enums not containing references). More value types can be added to handle more complex mathematical objects necessary for cryptographic specifications, like probability distributions for post-quantum cryptography.

The second direction for improvement deals with the formal foundations of hacspec. The compilation from hacspec to F* and/or EasyCrypt could be formalized and proven correct, either on paper or mechanically. The nature of hacspec's embedding inside Rust could also be explored, either by evaluating a reference interpreter of the language on a subset of Rust's official test suite, or by formally proving that hacspec's semantics are a special case of a larger semantics like Oxide or Rustbelt.

Overall, hacspec's goal is fairly different from any of the existing Rust frontends for verification frameworks. The domain-specific language can be viewed as a language for writing contracts and specifications for Rust functions within Rust, going beyond simple annotations. A third possible direction for future work would be to explore hacspec's relevance outside the world of cryptography, e.g. in systems programming.

Open Source Development

hacspec is being collaboratively developed on an open GitHub repository: https://github.com/hacspec/hacspec. The repository contains:

- the Rust source code of the hacspec typechecker and compilers to F* and EasyCrypt (see §2.3);
- the Rust source code of the hacspec libraries (see §3);
- a library of cryptographic specifications written in hacspec;
- the F* source code compiled from these specifications (see §4.1);
- Rust bindings for the Evercrypt [42] API provided by HACL*;

The Rust code is self-documented using rustdoc but various readmes and Makefiles are provided to guide developers on correctly using the toolchain.

References

[1] Al Fardan, N. J., and Paterson, K. G. Lucky thirteen: Breaking the tls and dtls record protocols. In *IEEE Symposium on Security and Privacy* (2013), p. 526–540.

[2] Almeida, J. B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., and Strub, P.-Y. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 1807–1823.

[3] Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. Verifying constant-time implementations. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016* (2016), T. Holz and S. Savage, Eds., USENIX Association, pp. 53–70.

[4] Almeida, J. B., Barbosa, M., Barthe, G., Grégoire, B., Koutsos, A., Laporte, V., Oliveira, T., and Strub, P.-Y. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), IEEE, pp. 965–982.

[5] Astrauskas, V., Müller, P., Poli, F., and Summers, A. J. Leveraging rust types for modular specification and verification. *Proc. ACM Program. Lang. 3*, OOPSLA (Oct. 2019).

[6] Balasubramanian, A., Baranowski, M. S., Burtsev, A., Panda, A., Rakamaric, Z., and Ryzhyk, L. System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017* (2017), A. Fedorova, A. Warfield, I. Beschastnikh, and R. Agarwal, Eds., ACM, pp. 156–161.

[7] BARBOSA, M., BARTHE, G., BHARGAVAN, K., BLANCHET, B., CREMERS, C., LIAO, K., AND PARNO, B. Sok: Computer-aided cryptography. In *IEEE Symposium on Security and Privacy (S&P'21)* (2021).

[8] BARTHE, G., BLAZY, S., GRÉGOIRE, B., HUTIN, R., LAPORTE, V., PICHARDIE, D., AND TRIEU, A. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang. 4*, POPL (2020), 7:1–7:30.

[9] BARTHE, G., DUPRESSOIR, F., GRÉGOIRE, B., KUNZ, C., SCHMIDT, B., AND STRUB, P.-Y. Easycrypt: A tutorial. In *Foundations of security analysis and design vii*. Springer, 2013, pp. 146–166.

[10] BERTOT, Y., AND CASTÉRAN, P. Le coq'art (v8). *Online http://wwwsop. inria. fr/members/Yves. Bertot/coqartF. pdf* (2015).

[11] BEURDOUCHE, B., BHARGAVAN, K., DELIGNAT-LAVAUD, A., FOURNET, C., KOHLWEISS, M., PIRONTI, A., STRUB, P.-Y., AND ZINZINDOHOUE, J. K. A messy state of the union: Taming the composite state machines of tls. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 535–552.

[12] BHARGAVAN, K., BLANCHET, B., AND KOBEISSI, N. Verified models and reference implementations for the tls 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 483–502.

[13] BHARGAVAN, K., KIEFER, F., AND STRUB, P.-Y. hacspec: Towards verifiable crypto standards. In *Security Standardisation Research* (Cham, 2018), C. Cremers and A. Lehmann, Eds., Springer International Publishing, pp. 1–20.

[14] BHARGAVAN, K., KOBEISSI, N., AND BLANCHET, B. Proscript tls: Building a tls 1.3 implementation with a verifiable protocol model. In *TRON Workshop-TLS 1.3, Ready Or Not* (2016).

[15] BJØRNER, N., GURFINKEL, A., MCMILLAN, K., AND RYBALCHENKO, A. *Horn Clause Solvers for Program Verification*. Springer International Publishing, Cham, 2015, pp. 24–51.

[16] BLANCHET, B. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.* (2001), IEEE, pp. 82–96.

[17] BLANCHET, B. Cryptoverif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied* (2007), vol. 117, p. 156.

[18] BÖCK, H., ZAUNER, A., DEVLIN, S., SOMOROVSKY, J., AND JOVANOVIC, P. Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS. In *USENIX Workshop on Offensive Technologies (WOOT)* (2016).

[19] BOND, B., HAWBLITZEL, C., KAPRITSOS, M., LEINO, K. R. M., LORCH, J. R., PARNO, B., RANE, A., SETTY, S., AND THOMPSON, L. Vale: Verifying high-performance cryptographic assembly code. In *26th {USENIX} Security Symposium ({USENIX} Security 17)* (2017), pp. 917–934.

[20] BRUMLEY, B. B., BARBOSA, M., PAGE, D., AND VERCAUTEREN, F. Practical realisation and elimination of an ecc-related software bug attack. In *Topics in Cryptology–CT-RSA 2012*. Springer, 2012, pp. 171–186.

[21] CADAR, C., DUNBAR, D., ENGLER, D. R., ET AL. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008), vol. 8, pp. 209–224.

[22] CREMERS, C., AND JACKSON, D. Prime, order please! revisiting small subgroup and invalid curve attacks on protocols using diffie-hellman. In *IEEE Computer Security Foundations Symposium (CSF)* (2019), pp. 78–93.

[23] DE MOURA, L., KONG, S., AVIGAD, J., VAN DOORN, F., AND VON RAUMER, J. The lean theorem prover (system description). In *International Conference on Automated Deduction* (2015), Springer, pp. 378–388.

[24] DELIGNAT-LAVAUD, A., FOURNET, C., KOHLWEISS, M., PROTZENKO, J., RASTOGI, A., SWAMY, N., ZANELLA-BÉGUELIN, S., BHARGAVAN, K., PAN, J., AND ZINZINDOHOUE, J. K. Implementing and proving the tls 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 463–482.

[25] DENIS, X. Deductive program verification for a language with a Rust-like typing discipline. Intership report, Université de Paris, Sept. 2020.

[26] DOCKINS, R., FOLTZER, A., HENDRIX, J., HUFFMAN, B., MCNAMEE, D., AND TOMB, A. Constructing semantic models of programs with the software analysis workbench. In *Working Conference on Verified Software: Theories, Tools, and Experiments* (2016), Springer, pp. 56–72.

[27] DURUMERIC, Z., LI, F., KASTEN, J., AMANN, J., BEEKMAN, J., PAYER, M., WEAVER, N., ADRIAN, D., PAXSON, V., BAILEY, M., ET AL. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference* (2014), pp. 475–488.

[28] ERBSEN, A., PHILIPOOM, J., GROSS, J., SLOAN, R., AND CHLIPALA, A. Systematic generation of fast elliptic curve cryptography implementations. Tech. rep., Technical Report. https://people. csail. mit. edu/jgross/personal-website ..., 2018.

[29] ERBSEN, A., PHILIPOOM, J., GROSS, J., SLOAN, R., AND CHLIPALA, A. Simple high-level code for cryptographic arithmetic-with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 1202–1219.

[30] ERKÖK, L., AND MATTHEWS, J. Pragmatic equivalence and safety checking in cryptol. In *Proceedings of the 3rd workshop on Programming languages meets program verification* (2009), pp. 73–82.

[31] FILLIÂTRE, J.-C., AND PASKEVICH, A. Why3—where programs meet provers. In *European symposium on programming* (2013), Springer, pp. 125–128.

[32] FROMHERZ, A., GIANNARAKIS, N., HAWBLITZEL, C., PARNO, B., RASTOGI, A., AND SWAMY, N. A verified, efficient embedding of a verifiable assembly language. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 1–30.

[33] GALOIS, INC. crux-mir.

[34] JUNG, R., JOURDAN, J.-H., KREBBERS, R., AND DREYER, D. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages 2*, POPL (2017), 1–34.

[35] LINDNER, M., APARICIUS, J., AND LINDGREN, P. No panic! verification of rust programs by symbolic execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)* (2018), pp. 108–114.

[36] MATSUSHITA, Y., TSUKADA, T., AND KOBAYASHI, N. Rusthorn: Chc-based verification for rust programs. In *European Symposium on Programming* (2020), Springer, Cham, pp. 484–514.

[37] MERCADIER, D., AND DAGAND, P.-É. Usuba: high-throughput and constant-time ciphers, by construction. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019), pp. 157–173.

[38] MÜLLER, P., SCHWERHOFF, M., AND SUMMERS, A. J. Viper: A verification infrastructure for permission-based reasoning. In *International Conference on Verification, Model Checking, and Abstract Interpretation* (2016), Springer, pp. 41–62.

[39] NIR, Y., AND LANGLEY, A. Chacha20 and poly1305 for IETF protocols. *RFC 8439* (2018), 1–46.

[40] OPENSSL. ChaCha20/Poly1305 heap-buffer-overflow. CVE-2016-7054, 2016.

[41] PROTZENKO, J., BEURDOUCHE, B., MERIGOUX, D., AND BHARGAVAN, K. Formally verified cryptographic web applications in webassembly. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 1256–1274.

[42] PROTZENKO, J., PARNO, B., FROMHERZ, A., HAWBLITZEL, C., POLUBELOVA, M., BHARGAVAN, K., BEURDOUCHE, B., CHOI, J., DELIGNAT-LAVAUD, A., FOURNET, C., ET AL. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *IEEE Symposium on Security and Privacy (SP)* (2020), pp. 634–653.

[43] RAKAMARIC, Z., AND EMMI, M. SMACK: decoupling source language details from verifier implementations. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings* (2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 106–113.

[44] RAMANANANDRO, T., DELIGNAT-LAVAUD, A., FOURNET, C., SWAMY, N., CHAJED, T., KOBEISSI, N., AND PROTZENKO, J. Everparse: Verified secure zero-copy parsers for authenticated message formats. In *28th {USENIX} Security Symposium ({USENIX} Security 19)* (2019), pp. 1465–1482.

[45] SWAMY, N., HRITCU, C., KELLER, C., RASTOGI, A., DELIGNAT-LAVAUD, A., FOREST, S., BHARGAVAN, K., FOURNET, C., STRUB, P.-Y., KOHLWEISS, M., ZINZINDOHOUÉ, J.-K., AND ZANELLA-BÉGUELIN, S. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (Jan. 2016), ACM, pp. 256–270.

[46] ULLRICH, S. Simple verification of rust programs via functional purification, 2016.

[47] WEISS, A., PATTERSON, D., MATSAKIS, N. D., AND AHMED, A. Oxide: The essence of rust. *CoRR abs/1903.00982* (2019).

[48] ZINZINDOHOUÉ, J.-K., BHARGAVAN, K., PROTZENKO, J., AND BEURDOUCHE, B. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 1789–1806.

Appendix
Full hacspec formalization

The hacspec language, being embedded in Rust, shares with it some linearity properties of its type system. However, we severely restrict the borrowing feature: while retaining enough expressive power to write cryptographic specifications, the language behaves much simpler.

The typing judgment presented here does not cover all the features of the hacspec implementation. It focuses on the core of the language, its interesting type system and execution semantics. More precisely, we did not include the following elements :

- anything that can easily be desugared like assignment operators (`+=`, etc);
- the `if` expression, that coexists with the `if` statement;
- all the various types of integers, the casts between them and all the operators to manipulate them (we only have basic `int` type with arithmetic);
- type inference;
- polymorphism and traits, as we assume that the Rust compiler does the job of monomorphizing everything for us.

Typing

Contrasting with the operational semantics, the typing rules are fairly complex and restrictive. The reason for this complexity is our objective to stick to Rust behavior.

The typing environnement of hacspec is fairly standard. We need a type dictionary to enforce the named type discipline of Rust that covers the types declared by the `array!` macro. Please note that the contexts Γ and Δ are considered as unordered maps rather than ordered associative lists. As such, the rules have the relevant elements appear at the end or the beginning of those contexts without loss of generality.

$$
\begin{array}{lrcl}
\text{Typing context} & \Gamma & ::= & \varnothing \\
\text{(unordered map)} & & | & x : \tau, \Gamma \\
& & | & f : (\ [\tau]^+\ ) \to \mu, \Gamma \\
\text{Type dictionary} & \Delta & ::= & \varnothing \\
\text{(unordered map)} & & | & t \to [\ \mu; n \in \mathbb{N}\ ], \Delta
\end{array}
$$

The restrictions on borrowing lead to severe limitations on how we can manipulate values of linear type in our language, rendering it quite useless at first sight. Indeed, when you receive a reference as a function argument, you can only use it in expressions and perform identity let bindings with it. You cannot store it in memory or in a tuple and pass it around indirectly in your program. This behavior is well-suited for input and output buffers in cryptographic code.

However, in the spirit of Rust, we introduce an escape hatch from linearity under the form of the `Copy` trait implementation. This trait, that is primitive to the Rust language, is used to distinguish the values that are "cheap" to copy. This concerns all the reference-free μ types except `Seq`, whose size is not known at compilation time (and thus can be arbitrarily large). Paradoxically, `array!` types that can be as large as their `Seq` counterparts do benefit from the `Copy` trait; we replicate here the behavior of Rust. Indeed, because the length is known at compilation time, the code generation backend of Rust (LLVM) can optimize the representation of the array in memory, especially if the size is small.

With this setup, both `array!` and `Seq` represent a table of data. The moral difference between them is that `array!` is a table passed by value, whereas `Seq` is a tabled passed by reference (immutable). In the cryptographic specifications, `Seq` is mostly used for input and output messages, whose length is known only at runtime. Fixed-size chunks or blocks of data are rather implemented using `array!`.

$$\frac{}{\text{Implementing the } \texttt{Copy} \text{ trait} \quad \Delta \vdash \tau : \texttt{Copy}}$$

$$\textsc{CopyUnit}\ \frac{}{\Delta \vdash \texttt{unit} : \texttt{Copy}} \qquad \textsc{CopyBool}\ \frac{}{\Delta \vdash \texttt{bool} : \texttt{Copy}}$$

$$\textsc{CopyInt}\ \frac{}{\Delta \vdash \texttt{int} : \texttt{Copy}}$$

$$\textsc{CopyTuple}\ \frac{\Delta \vdash \tau_1 : \texttt{Copy} \quad \cdots \quad \Delta \vdash \tau_n : \texttt{Copy}}{\Delta \vdash (\ \tau_1, \ldots, \tau_n\ ) : \texttt{Copy}}$$

$$\textsc{CopyArray}\ \frac{\Delta \vdash \mu : \texttt{Copy}}{t \to [\ \mu; n\ ], \Delta \vdash t : \texttt{Copy}}$$

Because Rust has an affine type system, hacspec also enjoys an affine typing context with associated splitting rules (SPLITLINEAR). Please note that immutable references values can be duplicated freely in the context (SPLITDUPLICABLE). During an elaboration phase inside the Rust compiler, the linearity of the type system gets circumvented for `Copy` values with the insertion of `clone()` functions call that perform a copy of the value wherever the linear type system forces a copy of the value to be made. We formalize this behavior here by allowing `Copy` types duplication in the typing context, like immutable references (SPLITCOPY). Lastly, functions are always duplicable in

the context (SplitFunction). In the following, Γ behaves like an unordered map from variables to their types.

Context splitting	$\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$
Borrowed context splitting	$\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$

SplitEmpty
$$\frac{}{\Delta \vdash \varnothing = \varnothing \circ \varnothing}$$

SplitLinear1
$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2}{\Delta \vdash x : \tau, \Gamma = (\, x : \tau, \Gamma_1 \,) \circ \Gamma_2}$$

SplitLinear2
$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2}{\Delta \vdash x : \tau, \Gamma = \Gamma_1 \circ (\, x : \tau, \Gamma_2 \,)}$$

SplitDuplicable
$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2}{\Delta \vdash x : \&\tau, \Gamma = (\, x : \&\tau, \Gamma_1 \,) \circ (\, x : \&\tau, \Gamma_2 \,)}$$

SplitCopy
$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Delta \vdash \tau : \texttt{Copy}}{\Delta \vdash x : \tau, \Gamma = (\, x : \tau, \Gamma_1 \,) \circ (\, x : \tau, \Gamma_2 \,)}$$

SplitFunction
$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2}{\begin{array}{l}\Delta \vdash f : (\, \mu_1, \ldots, \mu_n \,) \to \tau, \Gamma = \\ \quad (\, f : (\, \mu_1, \ldots, \mu_n \,) \to \tau, \Gamma_1 \,) \circ \\ \quad (\, f : (\, \mu_1, \ldots, \mu_n \,) \to \tau, \Gamma_2 \,)\end{array}}$$

We can now proceed to the main typing judgments. TypVarLinear and TypVarDup reflect the variable typing present in the context. TypTupleCons only allows non-reference values inside a tuple, with a linear context splitting to check each term of the tuple. TypArrayAccess, TypSeqAccess and TypSeqRefAccess specify the array indexing syntax, which is overloaded to work with both `array!`, `Seq` and `&Seq`. This corresponds to the implementing of the `Index` trait in Rust.

The function call rule, TypFuncCall, is the most complex rule of the typing judgment, because it contains the restricted borrowing form allowed in hacspec. First, note that the context is split for typechecking the arguments of the function, because a linear value cannot be used in two arguments. Next, TypFunArg ensures that the arguments are well-typed. However, if an argument is borrowed at call-site, then TypFunArgBorrow and TypFunArgBorrowVar checks the value that is being borrowed under the reference. In our degenerate pattern of borrowing, TypFunArgBorrow and TypFunArgBorrowVar are doing the work of the Rust borrow checker. The last rules for binary and unary operations are standard.

Value typing	$\Gamma; \Delta \vdash v : \mu$
Expression typing	$\Gamma; \Delta \vdash e : \tau \Rrightarrow \Gamma'$
Function argument typing	$\Gamma; \Delta \vdash a \sim \tau \Rrightarrow \Gamma'$

TypUnit
$$\frac{}{\Gamma; \Delta \vdash \texttt{()} : \texttt{unit}}$$

TypBool
$$\frac{b \in \{\texttt{true}, \texttt{false}\}}{\Gamma; \Delta \vdash b : \texttt{bool}}$$

TypInt
$$\frac{n \in \mathbb{N}}{\Gamma; \Delta \vdash n : \texttt{int}}$$

TypSeqValue
$$\frac{\forall i \in [\![1, n]\!],\ \Gamma; \Delta \vdash v_i : \mu}{\Gamma; \Delta \vdash [\, v_1, \ldots, v_n \,] : \texttt{Seq}\langle\, \mu \,\rangle}$$

TypArrayValue
$$\frac{\forall i \in [\![1, n]\!],\ \Gamma; \Delta \vdash v_i : \mu}{\Gamma; \Delta \vdash [\, v_1, \ldots, v_n \,] : [\, \mu; n \,]}$$

TypValueAsExpr
$$\frac{\Gamma; \Delta \vdash v : \mu}{\Gamma; \Delta \vdash v : \mu \Rrightarrow \Gamma}$$

TypVarLinear
$$\frac{}{x : \tau, \Gamma; \Delta \vdash x : \tau \Rrightarrow \Gamma}$$

TypVarDup
$$\frac{x : \&\tau \in \Gamma}{\Gamma; \Delta \vdash x : \&\tau \Rrightarrow \Gamma}$$

TypTupleElim
$$\frac{\Gamma; \Delta \vdash e : (\, \mu_1, \ldots, \mu_m \,) \Rrightarrow \Gamma' \qquad n \in [\![1, m]\!]}{\Gamma; \Delta \vdash e.n : \mu_n \Rrightarrow \Gamma'}$$

TypTupleRefElim
$$\frac{\Gamma; \Delta \vdash e : \&(\, \mu_1, \ldots, \mu_m \,) \Rrightarrow \Gamma' \qquad n \in [\![1, m]\!]}{\Gamma; \Delta \vdash e.n : \&\mu_n \Rrightarrow \Gamma'}$$

TypTupleIntro
$$\frac{\begin{array}{c}\Delta \vdash \Gamma = \Gamma_1 \circ \cdots \circ \Gamma_n \\ \forall i \in [\![1, n]\!],\ \Gamma_i; \Delta \vdash e_i : \mu_i \Rrightarrow \Gamma'_i \\ \Delta \vdash \Gamma' = \Gamma'_1 \circ \cdots \circ \Gamma'_n\end{array}}{\Gamma; \Delta \vdash (\, e_1, \ldots, e_n \,) : (\, \mu_1, \ldots, \mu_n \,) \Rrightarrow \Gamma'}$$

TypArrayAccess
$$\frac{\begin{array}{c}t \to [\, \mu; n \,] \in \Delta \\ \Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1; \Delta \vdash x : t \Rrightarrow \Gamma'_1 \\ \Gamma_2; \Delta \vdash e : \texttt{int} \Rrightarrow \Gamma'_2 \qquad \Delta \vdash \Gamma' = \Gamma'_1 \circ \Gamma'_2\end{array}}{\Gamma; \Delta \vdash x[\, e \,] : \mu \Rrightarrow \Gamma'}$$

TypSeqAccess
$$\frac{\begin{array}{c}\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1; \Delta \vdash x : \texttt{Seq}\langle\, \mu \,\rangle \Rrightarrow \Gamma'_1 \\ \Gamma_2; \Delta \vdash e : \texttt{int} \Rrightarrow \Gamma'_2 \qquad \Delta \vdash \Gamma' = \Gamma'_1 \circ \Gamma'_2\end{array}}{\Gamma; \Delta \vdash x[\, e \,] : \mu \Rrightarrow \Gamma'}$$

TypSeqRefAccess
$$\frac{\begin{array}{c}\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1; \Delta \vdash x : \&\texttt{Seq}\langle\, \mu \,\rangle \Rrightarrow \Gamma'_1 \\ \Gamma_2; \Delta \vdash e : \texttt{int} \Rrightarrow \Gamma'_2 \qquad \Delta \vdash \Gamma' = \Gamma'_1 \circ \Gamma'_2\end{array}}{\Gamma; \Delta \vdash x[\, e \,] : \mu \Rrightarrow \Gamma'}$$

TypFuncCall
$$\frac{\begin{array}{c}f : (\, \mu_1, \ldots, \mu_n \,) \to \tau \in \Gamma \\ \Delta \vdash \Gamma = \Gamma_1 \circ \cdots \circ \Gamma_n \\ \forall i \in [\![1, n]\!],\ \Gamma_i; \Delta \vdash a_i \sim \mu_i \Rrightarrow \Gamma'_i \\ \Delta \vdash \Gamma' = \Gamma'_1 \circ \cdots \circ \Gamma'_n\end{array}}{\Gamma; \Delta \vdash f(\, a_1, \ldots, a_n \,) : \tau \Rrightarrow \Gamma'}$$

TypFunArg
$$\frac{\Gamma; \Delta \vdash e : \tau \Rrightarrow \Gamma'}{\Gamma; \Delta \vdash e \sim \tau \Rrightarrow \Gamma'}$$

TypFunArgBorrow
$$\frac{\Gamma; \Delta \vdash e : \mu \Rrightarrow \Gamma'}{\Gamma; \Delta \vdash \&e \sim \&\mu \Rrightarrow \Gamma'}$$

TypFunArgBorrowVar
$$\frac{\Gamma; \Delta \vdash x : \mu \Rrightarrow \_}{\Gamma; \Delta \vdash \&x \sim \&\mu \Rrightarrow \Gamma}$$

TypBinopInt
$$\frac{\begin{array}{c}\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \\ \Gamma_1; \Delta \vdash e_1 : \texttt{int} \Rrightarrow \Gamma'_1 \qquad \Gamma_2; \Delta \vdash e_2 : \texttt{int} \Rrightarrow \Gamma'_2 \\ \odot \in \{\, +, -, *, / \,\} \qquad \Delta \vdash \Gamma' = \Gamma'_1 \circ \Gamma'_2\end{array}}{\Gamma; \Delta \vdash e_1 \ \odot\ e_2 : \texttt{int} \Rrightarrow \Gamma'}$$

TypBinopBool

$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \quad \Gamma_1; \Delta \vdash e_1 : \texttt{bool} \Rrightarrow \Gamma_1' \quad \Gamma_2; \Delta \vdash e_2 : \texttt{bool} \Rrightarrow \Gamma_2' \quad \odot \in \{ \texttt{\&\&}, \texttt{||} \} \quad \Delta \vdash \Gamma' = \Gamma_1' \circ \Gamma_2'}{\Gamma; \Delta \vdash e_1 \ \odot \ e_2 : \texttt{bool} \Rrightarrow \Gamma'}$$

TypBinopComp

$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \quad \Gamma_1; \Delta \vdash e_1 : \texttt{int} \Rrightarrow \Gamma_1' \quad \Gamma_2; \Delta \vdash e_2 : \texttt{int} \Rrightarrow \Gamma_2' \quad \odot \in \{ \texttt{==}, \texttt{!=}, \texttt{>}, \texttt{<} \} \quad \Delta \vdash \Gamma' = \Gamma_1' \circ \Gamma_2'}{\Gamma; \Delta \vdash e_1 \ \odot \ e_2 : \texttt{bool} \Rrightarrow \Gamma'}$$

TypUnopInt

$$\frac{\Gamma; \Delta \vdash e : \texttt{int} \Rrightarrow \Gamma' \quad \oslash \in \{ \texttt{-} \}}{\Gamma; \Delta \vdash \oslash \ e : \texttt{int} \Rrightarrow \Gamma'}$$

TypUnopBool

$$\frac{\Gamma; \Delta \vdash e : \texttt{bool} \Rrightarrow \Gamma' \quad \oslash \in \{ \texttt{!} \}}{\Gamma; \Delta \vdash \oslash \ e : \texttt{bool} \Rrightarrow \Gamma'}$$

Let's now move to the statement typing. We've chosen statements here rather than nested expressions because of the Rust behavior of the `if` statement and the `for` loop. A list of statement corresponds to a block, introduced in Rust by $\{ \cdots \}$. The single statement typing judgment produces a new Γ' because of variable definitions inside a block. Single statements also yield back a type, because the last statement of the function is also the return value of the function. All statement type except the last one should be `unit`.

The rule TypLet introduces a new mutable local variable, that can later be reassigned (TypReassign) in the program. In Rust, the `mut` indicates that the local variable is mutable, in its absence, variable reassignments are prohibited. In this formalization, all variables are mutable for simplification. The main use of mutable local variables is for variables that are mutated inside a `for` loop. Indeed, because `for` loops are restricted to integer range iteration, we cannot express what would normally be a fold without these mutable variables. Because the mutable variables are local to a block, we do not need to formalize a full-fledged heap for the operational semantics. Rather, we will model them as a limited piece of state that gets passed around during execution.

Next, TypArrayAssign and TypSeqAssign define the overloading of the array update syntax that works for both `array!` and `Seq`. Note that TypIfThenElse use the same context Γ for the two branches of the conditional.

Statement typing	$\Gamma; \Delta \vdash s : \tau \Rrightarrow \Gamma'$
Block typing	$\Gamma; \Delta \vdash b : \tau \Rrightarrow \Gamma'$

TypLet

$$\frac{x \notin \Gamma \quad \Gamma; \Delta \vdash e : \tau \Rrightarrow \Gamma'}{\Gamma; \Delta \vdash \texttt{let} \ x : \tau = e : \texttt{unit} \Rrightarrow \Gamma', x : \tau}$$

TypReassign

$$\frac{x : \tau \in \Gamma \quad \Gamma; \Delta \vdash e : \tau \Rrightarrow \Gamma'}{\Gamma; \Delta \vdash x = e : \texttt{unit} \Rrightarrow \Gamma', x : \tau}$$

TypArrayAssign

$$\frac{x : t \in \Gamma \quad t \to [\ \mu; n\ ] \in \Delta \quad \Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \quad \Gamma_1; \Delta \vdash e_1 : \texttt{int} \Rrightarrow \Gamma_1' \quad \Gamma_2; \Delta \vdash e_2 : \mu \Rrightarrow \Gamma_2' \quad \Delta \vdash \Gamma' = \Gamma_1' \circ \Gamma_2'}{\Gamma; \Delta \vdash x\texttt{[}\ e_1\ \texttt{]} = e_2 : \texttt{unit} \Rrightarrow x : t, \Gamma'}$$

TypSeqAssign

$$\frac{x : \texttt{Seq<}\ \mu\ \texttt{>} \in \Gamma \quad \Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \quad \Gamma; \Delta \vdash e_1 : \texttt{int} \Rrightarrow \Gamma_1' \quad \Gamma; \Delta \vdash e_2 : \mu \Rrightarrow \Gamma_2' \quad \Delta \vdash \Gamma' = \Gamma_1' \circ \Gamma_2'}{\Gamma; \Delta \vdash x\texttt{[}\ e_1\ \texttt{]} = e_2 : \texttt{unit} \Rrightarrow x : \texttt{Seq<}\ \mu\ \texttt{>}, \Gamma'}$$

TypIfThen

$$\frac{\Gamma; \Delta \vdash e : \texttt{bool} \Rrightarrow \Gamma' \quad \Gamma'; \Delta \vdash b : \texttt{unit} \Rrightarrow \Gamma''}{\Gamma; \Delta \vdash \texttt{if}\ e\ \texttt{then}\ b : \texttt{unit} \Rrightarrow \Gamma''}$$

TypIfThenElse

$$\frac{\Gamma; \Delta \vdash e : \texttt{bool} \Rrightarrow \Gamma_c \quad \Gamma_c; \Delta \vdash b : \texttt{unit} \Rrightarrow \Gamma_t \quad \Gamma_c; \Delta \vdash b' : \texttt{unit} \Rrightarrow \Gamma_f \quad \Gamma' = \Gamma \cap \Gamma_f \cap \Gamma_t}{\Gamma; \Delta \vdash \texttt{if}\ e\ \texttt{then}\ b\ \texttt{else}\ b' : \texttt{unit} \Rrightarrow \Gamma'}$$

TypForLoop

$$\frac{\Gamma; \Delta \vdash e_1 : \texttt{int} \Rrightarrow \Gamma_1 \quad \Gamma_1; \Delta \vdash e_2 : \texttt{int} \Rrightarrow \Gamma_2 \quad \Gamma_2, x : \texttt{int}; \Delta \vdash b : \texttt{unit} \Rrightarrow \Gamma_b \quad \Gamma_2 \subset \Gamma_b \quad \Gamma' = \Gamma \cap \Gamma_b}{\Gamma; \Delta \vdash \texttt{for}\ x\ \texttt{in}\ e_1\ \texttt{..}\ e_2\ b : \texttt{unit} \Rrightarrow \Gamma'}$$

TypBlock

$$\frac{\Gamma; \Delta \vdash s_1 : \texttt{unit} \Rrightarrow \Gamma' \quad \Gamma'; \Delta \vdash \{ s_2; \ldots; s_n \} : \tau}{\Gamma; \Delta \vdash \{ s_1; \ldots; s_n \} : \tau}$$

TypBlockOne

$$\frac{\Gamma; \Delta \vdash s_1 : \tau \Rrightarrow \Gamma'}{\Gamma; \Delta \vdash \{ s_1 \} : \tau}$$

TypBlockAsStatement

$$\frac{\Gamma; \Delta \vdash b : \tau}{\Gamma; \Delta \vdash b : \tau \Rrightarrow \Gamma}$$

TypExpToStmt

$$\frac{\Gamma; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash e : \tau \Rrightarrow \Gamma}$$

A hacspec program is a list of items i. Their typing judgment produces both a new Γ and Δ, because an item introduces either a new function or a new named type. Please note, as mentionned before, that the return type of functions is restricted to μ, as returning a reference is forbidden. The TypFnDecl also means that recursion is forbidden in hacspec, since f is not passed in its typing context.

Item typing	$\Gamma; \Delta \vdash i \Rrightarrow \Gamma'; \Delta'$

TypArrayDecl

$$\frac{}{\Gamma; \Delta \vdash \texttt{array!}\ (\ t, \ \mu, \ n\ ) \Rrightarrow \Gamma; \Delta, t \to [\ \mu; n\ ]}$$

TypFnDecl

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n; \Delta \vdash b : \mu}{\Gamma; \Delta \vdash \texttt{fn}\ f\ (\ x_1 : \tau_1, \ldots, x_n : \tau_n\ )\ \texttt{->}\ \mu\ b \Rrightarrow \Gamma, f : (\ \tau_1, \ldots, \tau_n\ ) \to \mu; \Delta}$$