hacspec: Towards Verifiable Crypto Standards

Karthikeyan Bhargavan, Franziskus Kiefer, Pierre-Yves Strub

hacspec: towards verifiable crypto standards

Karthikeyan Bhargavan[1], Franziskus Kiefer[2], and Pierre-Yves Strub[3]

[1] INRIA, Paris (karthikeyan.bhargavan@inria.fr)
[2] Mozilla (mail@franziskuskiefer.de)
[3] École Polytechnique (pierre-yves@strub.nu)

Abstract. We present hacspec, a collaborative effort to design a formal specification language for cryptographic primitives. Specifications (specs) written in hacspec are succinct, easy to read and implement, and lend themselves to formal verification using a variety of existing tools. The syntax of hacspec is similar to the pseudocode used in cryptographic standards but is equipped with a static type system and syntax checking tools that can find errors. Specs written in hacspec are executable and can hence be tested against test vectors taken from standards and specified in a common format. Finally, hacspec is designed to be compilable to other formal specification languages like F*, EasyCrypt, Coq, and cryptol, so that it can be used as the basis for formal proofs of functional correctness and cryptographic security using various verification frameworks. This paper presents the syntax, design, and tool architecture of hacspec. We demonstrate the use of the language to specify popular cryptographic algorithms, and describe preliminary compilers from hacspec to F* and to EasyCrypt. Our goal is to invite authors of cryptographic standards to write their pseudocode in hacspec and to help the formal verification community develop the language and tools that are needed to promote high-assurance cryptographic sofware backed by mathematical proofs.

1 Introduction

Cryptographic algorithms such as those standardized by organizations like the NIST, IETF, and the ISO, are written with a focus on clarity, ease of implementation, and interoperability. To this end, standards authors employ a combination of carefully edited text, precise mathematical formulas, and pseudocode. Many standards include test vectors and some even include reference code written in popular languages like C or python. Each standard is looked over by dozens of experts before publication and then closely vetted by the developer community during implementation and deployment.

Standards vs. Implemenations. Despite this care, implementing cryptographic standards remains challenging and error-prone for a variety of reasons.

First, the mathematical structure used in a cryptographic algorithm may be easy to specify but hard to implement correctly and efficiently. For example, most elliptic curve standards such as [2], require modular arithmetic over large

prime fields, but implementing bignum arithmetic efficiently and correctly is hard, leading to numerous subtle bugs that are hard to find simply by testing. [1]

Second, the specification of an algorithm in a standard may not account for side-channel attacks that require specific countermeasures to be implemented. For example, the natural way to implement the AES standard [14] leads to cache-timing attacks [7] that are hard to protect against without significantly changing the main routines of the algorithm [18].

Third, standards typically do not specify the interface (API) through which applications may use the cryptographic algorithm in practice. For example, the SHA-2 specification [11] specifies how a message can be hashed in one pass, but does not describe the incremental hashing interface that is commonly used, and which can be quite error-prone to implement [20].

Fourth, the security guarantees and assumptions of a cryptographic algorithm are often subtle and easy to misunderstand, leading to dangerous vulnerabilities. For example, the GCM [13] and ECDSA [16] specifications both require unique nonces but the use of bad randomness and incorrect configuration has led to real-world implementations that repeat nonces, leading to practical attacks [9,12].

These are just a few of the reasons why there is often a substantial gap between a cryptographic standard and its implementations. It is hard to close this gap just with testing since cryptographic algorithms often have corner cases that are reached only with low probability and some properties like side-channel resistance and cryptographic security are particularly hard to test. Instead, we advocate the use of formal methods to mathematically prove that an implementation meets the standard and protects against side channels as well as programmatically check expected cryptographic security guarantees of the spec.

Formal Verification of Cryptography. A wide range of verification frameworks have been proposed for the analysis of cryptographic algorithms and their implementations. The Software Analysis Workbench (SAW) can verify C and Java implementations for input-output equivalence with specifications written in Cryptol, a domain-specific language for specifying cryptographic primitives [24]. The Verified Software Toolchain can be used to verify C implementations against specifications written in Coq [4]. HACL* is a library of modern cryptographic algorithms that are written in F*, verified against specifications written in F*, and then compiled to C [25]. Fiat-Crypto generates efficient verified C code for field arithmetic from high-level specifications embedded in Coq [15]. Vale can be used to verify assembly implemenations of cryptography against specifications written in Dafny or F* [10]. Jasmin is another crypto-oriented assembly language verification tool with a verified compiler written in Coq [3]. EasyCrypt can be used to build cryptographic game-based security proofs for constructions and protocols written in a simple imperative language [6]. The Foundational Cryptography Framework (FCF) mechanizes proofs of cryptographic schemes in Coq [21]. CryptoVerif can be used to develop machine-checked proofs of cryptographic protocols written in the applied pi calculus [8].

[1] See, for example, this bug in a popular Curve25519 implementation `https://www.imperialviolet.org/2014/09/07/provers.html`

In order to benefit from any of these verification frameworks, the first step is to write a formal specification of the cryptographic algorithm. Since each framework uses its own specification language, this step can be tedious, time-consuming, and error-prone. Furthermore, the resulting formal specification is often tailored to suit the strengths of a particular verification framework, making it difficult to check the conformance of the specification with the standard or to compare it with other formal specifications.

Ideally, the published standard would itself include a formal-enough reference specification from which these tool-specific specs could be derived. Indeed, standards bodies have considered incorporating formal languages in the past. Since 2001, the IETF has a guideline on the use of formal languages in standards, which sets out sensible requirements, such as "The specification needs to be verifiable".[2] Nevertheless, the use of formal languages in cryptographic standards has not caught on, probably because authors did not see a significant benefit in exchange for the time and effort required to write formal specifications.

In this paper we propose hacspec, a new domain-specific formal language for cryptographic algorithms that we believe is suitable for use in standards. The language comes equipped with syntax and type checkers, a testing framework, and several verification tools. Hence, standards authors can use it to formalize, test, and verify pseudocode written in hacspec before including it in the standard. Furthermore, the language provides a common front-end for various cryptographic verification frameworks, so that proofs in these frameworks can be precisely compared and composed.

The authors believe that it is vital for a cryptographic standard to not only specify the mathematical algorithm describing the standard but also to allow engineers to implement the specification securely. In order to securely implement a specification it is important to make sure that the implementation is correct, i.e. that the implementation is functionally equivalent to the specification. This is especially important for highly optimized implementations that are hard to verify manually. The mechanisms proposed in this paper allow to prove this correctness property. The proposed tools further allow to prove cryptographic properties of the specified algorithm as well as security properties of an implementation. While these additional proofs do not necessarily belong into a specification of an algorithm, it makes the specification the single document of reference when implementing the algorithm or examining its security.

hacspec is designed in order to keep the barriers of entry low by being very close to what some specification authors use already and most engineers, mathematicians, and researchers are familiar with. Because of the design and additional benefits hacspec offers the authors believe that hacspec has a good chance to get adopted by specification authors.

Contributions and Outline The design of hacspec originates from discussions at the HACS workshop[3] held alongside the Real-World Crypto conference

[2] `https://www.ietf.org/iesg/statement/pseudocode-guidelines.html`
[3] `https://hacs-workshop.github.io/`

2018. The workshop participants included crypto developers for many major crypto libraries as well as researchers behind many popular verification frameworks. Together, we sought to achieve a balance between developer usability and ease of formal verification. This paper is the realization and continuation of the discussions and outcomes of that group.

We present hacspec in Section 2, a new specification and verification architecture for cryptographic standards. We describe the syntax of the hacspec language and show how it can be used to write cryptographic specifications on some examples in Section 3. We present compilers from hacspec to F* and EasyCrypt and show how the resulting specifications are used as verification targets in formal proofs in Section 5.1 and 5.2. Finally, we present our current status and ongoing work in Section 6. Everything described in this paper is implemented and available in the hacspec git repository.[4]

2 Architecture and Design Goals

Figure 1 depicts the proposed architecture of hacspec and design workflow. In the remainder of this section we describe each component and its design goals. In subsequent sections, we will describe in more detail how our current tool-set realizes these goals, and we will present preliminary results.

A new specification language. hacspec is a new domain-specific language for cryptographic algorithms that aims to build a bridge between standards authors and formal verification tools. Consequently, the language needs to be familiar to spec writers, mathematicians, and crypto developers alike, but with precise semantics that can be easily translated to various formal specification languages.

We chose to design hacspec as a subset of python (version 3.6.4) since python syntax is already used as pseudocode in various standards, e.g. [19,17], and hence is familiar to standards authors and crypto developers. Python is especially well suited for this task since it is an imperative language that supports native bignums (arbitrary-size integers) and has an expressive type annotation syntax. As we will see in Section 3 we restrict and enhance the python syntax in many ways to enable compilation to typed formal specifications.

Scope and Limitations. Although hacspec may eventually be used as a starting point for specifying cryptographic protocols and APIs, we emphasize that capturing anything other than cryptographic algorithms is out of scope for hacspec at this point. By focusing only on cryptographic algorithms, we believe we obtain a simpler syntax and are able to design more usable tools. Also note that the language as defined in Section 3 currently does not allow to use more advanced concepts such as higher-order functions. Because it would increase the complexity of the compiler, more advanced features — while planned — are left for future work.

Syntax and Type Checking. Since specifications written in hacspec are meant to be used as a basis for formal proofs, it is important that the specs themselves

[4] `https://github.com/hacs-workshop/hacspec/`

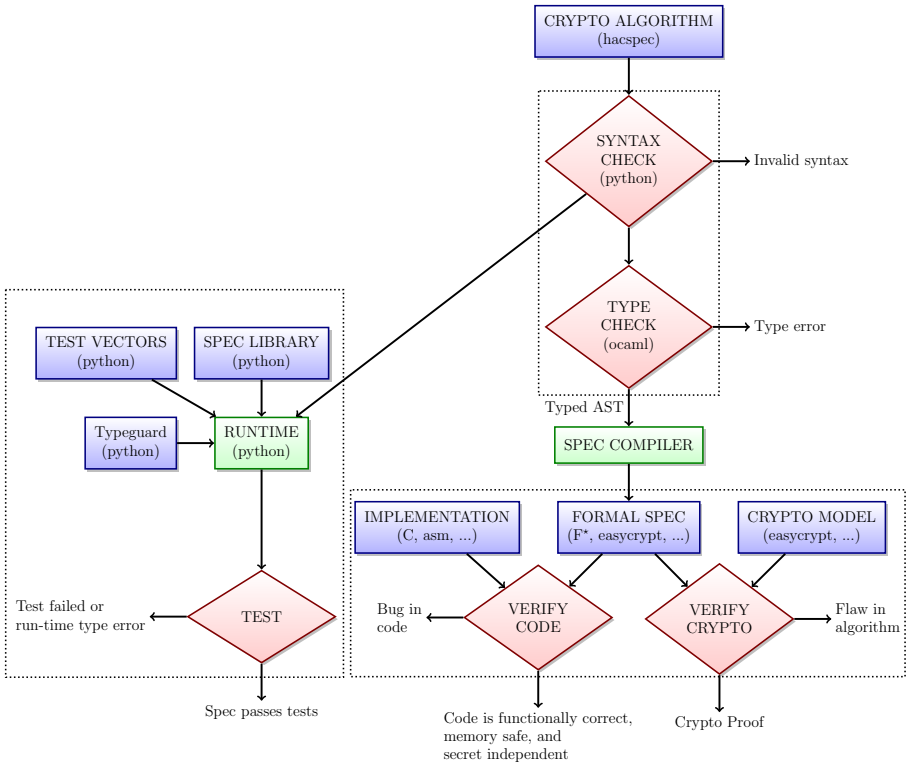

Fig. 1. hacspec specification and verification architecture. Authors of cryptographic standards write their alogrithm in hacspec, aided by the syntax and type checker. They can then test their specs by executing the spec in python (along with an implementation of the builtin library). They can also use spec compilers to generate formal specifications in languages like F*, EasyCrypt, etc., that can in turn be used as the basis for verifying the correctness of various implementations, or for proving the cryptographic security guarantees of the algorithm.

are correct. We propose several static and run-time tools that can find common programming errors in hacspec specifications.

First, we provide a syntax checker written in python that can find simple errors and reject specifications that use python features or control structures that are not supported by our domain-specific language. Second, we use a run-time type checker plugin for python called TypeGuard to find type errors in hacspec specifications. Using these two tools, authors who only have python on their system can already author and check hacspec specifications.

As a third tool, we also provide a static type checker for hacspec written in OCaml which can find violations of the annotated types at compile time. In addition to finding type errors, this typechecker generates a typed abstract syntax tree (AST) that is then used as a basis for our spec compilers.

Running & Testing Specifications. A key goal of hacspec is that the specifications should be executable. hacspec specifications are already written in a subset of python but they use many domain-specific types and data structures. We provide a library written in python that implements these builtin types and data structures, so that the specification can be run by a standard python interpreter and tested against test vectors.

Hence, a hacspec can also be seen as a reference implementation of the standard that can be used to guide other implementations and test them for correctness. However, these specs should clearly not be used in real-world deployments as they are not intended to be side-channel free or performant.

Verifying Implementations. The hacspec language is designed to support compilation to formal specifications for a variety of verification frameworks. We present a compiler from hacspec to F* in Section 5.1 and show that specifications generated by this compiler can be used to verify implementations written in Low*, a subset of F* that can be compiled to C.

We also envision compilers from hacspec to Cryptol and Coq models that can serve as formal specifications for verifying code written in C, Java, and assembly, using the SAW, Fiat-Crypto, VST, and Jasmin verification frameworks.

Verifying Cryptographic Security. Formal specifications compiled from hacspec can also be used as the basis for cryptographic proofs. We present a compiler from hacspec to EasyCrypt in Section 5.2 and show how such specifications can be used to prove that a construction meets its cryptographic security guarantees under precise assumptions on its underlying primitives. Hence, we intend for hacspec to provide a starting point for mechanized cryptographic proofs of standards. We also envision compilers that can generate specifications for other cryptographic verification frameworks like FCF [22].

3 The hacspec Language

In this section, we will describe the syntax of hacspec language. As we shall see in Section 3.2 this syntax is already adequate to write cryptographic algorithms. But for developer convenience we also provide a builtin library that contains a number of commonly-used types, constants, operators, and functions. This library is described in Section 3.3.

3.1 hacspec syntax

The full syntax of hacspec is depicted in Table 1, divided in terms of values, expressions, types, statements, and specs.

Values. Specifications in hacspec define computations over values. The simplest values are arbitrary-size integers, booleans, and strings. Integer constants can be written in decimal, binary, octal, or binary; booleans are `True` or `False`; string constants begin and end with quotes (`"` or `'`). The builtin library defines additional constants. The two main data structures used in hacspec are tuples and

Values v ::=	`n`	*integer constants*
	| `True` | `False`	*boolean constants*
	| `'...'` | `"..."`	*string constants*
	| `(v1,...,vn)`	*tuple constant*
	| `array([v1,...,vn])`	*array constant*
Expressions e ::=	`v`	*values*
	| `x` | `m.x`	*local and global variables*
	| `(e1,...,en)`	*tuple construction*
	| `array([e1,...,en])`	*array construction*
	| `array.length(e)`	*array length*
	| `e[e0]`	*array access*
	| `e[e0:e1]`	*array slice*
	| `e(e1,...,en)`	*function call*
	| `e1 binop e2`	*builtin binary operators*
	| `unaryop e`	*builtin unary operators*
Types t ::=	`int`, `str`, `bool`	*basic types*
	| `tuple_t(t1,...,tn)`	*tuples*
	| `vlarray_t(t)`	*variable-length array*
	| `x`	*user-defined or builtin type*
	| `x(t1,...,tn,e1,...,em)`	*builtin type application*
Statements s ::=	`x: Type = t`	*type declaration*
	| `x: t`	*variable declaration*
	| `x = e`	*variable assignment*
	| `x binop= e`	*augmented variable assignment*
	| `(x1,..,xn) = e`	*tuple matching*
	| `x[i] = e`	*array update*
	| `x[i] binop= e`	*augmented array update*
	| `x[i:j] = e`	*array slice update*

	| `if e:`
	`  s1...sn`
	`elif e:`
	`  s1'...sn'`
	`else`
	`  s1''...sn''`
		if-elif-else conditional

	| `for i in range(e):` `  s1...sn`	*for loop*
	| `break`	*break from loop*
	| `def x(x1:t1,...,xn:tn) → t:` `  s1 ... sn`	*function declaration*
	| `return e`	*return from function*
	| `from x import x1, x2,..., xn`	*module import*
Specs σ ::=	`s1...sn`	*sequence of statements*

Table 1. hacspec syntax: values, expressions, types, statements, and specifs. For types, constants, functions, and operators provided by the builtin library, see Section 3.3.

arrays. Tuple constants are written `(v1,...,vn)`, parentheses are optional, and array constants are written `array([v1,...,vn])`.

Expressions. Expressions represent purely functional computations that yield values without any side-effects. Expressions include values as well as tuples and arrays constructed with a sequence of expressions (evaluated left to right). Variables (`x`) may be local or global; global variables may be qualified by a module name (e.g. `array.empty`). Global variables may only be assigned once when they are initialized; thereafter they are read-only.

Arrays can be accessed in two ways: `a[i]` reads the `i`'th element of an array `a`; `a[i:j]` copies a slice of the array `a` from index `i` (inclusive) to index `j` (exclusive) into a fresh array and returns the result.

A function call, written `f(e1,...,en)`, calls the function `f` that may be user-defined or from the builtin library. A special case of builtin functions are unary and binary infix operators (e.g. arithmetic operators like `+`, bitwise operators like `<<`, and comparison operators like `==`).

Functions in hacspec are call-by-value, the arguments to the function are first evaluated, then the resulting values are *copied* into the arguments of the function, and then the function is called with these arguments. The function itself may allocate and modify local state, as we will see below, but all this state is deallocated when the function returns; only the output of the function remains. Consequently, function calls are *observationally pure*, i.e. they have no side effects and hence can be treated as pure functions that return a fresh result.

Notably, all hacspec expressions produce a fresh value: even when an expression evaluates to an array, the resulting value is a fresh array value, not a pointer into or an alias to some other array.

Types. The basic types in hacspec are integers (`int`), booleans (`bool`), and strings (`str`). Types can be parameterized, that is, they are defined as functions over both types and values; such parametric types can be instantiated by applying them to specific types and expressions `x(t1,...,tn,e1,...,em)`. Two special cases of such parametric types are tuples and arrays. Each n-tuple has a type `tuple_t(t1,...,tn)`, where the first component has type `t1`, the second has type `t2`, and so on. Arrays have type `vlarray_t(t)`, where the contents of the array have type `t`. These are called variable-length arrays since their length is only known at run-time.

As we will see, the builtin library defines more types and type constructors for writing refinement types, which can be used to annotate and check specs for more advanced logical properties.

Statements. A hacspec specification is a sequence of statements, each of which defines a global constant, a type, or a function. Such statement sequences are written one on each line (hence with a newline as a separator), with the indentation of each line indicating the block that the statement belongs to. This syntactic convention, taken from python, simplifies and unclutters the specification by removing the need for block and statement separators.

All variables must be declared before use; thus the statement `x:t` declares a new variable `x` that has type `t`, whereas the statement `x:t = e` declares and

initializes `x` that has type `t` to the value obtained by evaluating `e`. New type abbreviations are defined by writing `x:Type = t`; by convention all user-defined type names are suffixed by `_t` (e.g. `vlarray_t`).

Local variables can be modified with an assignment `x = e` or an augmented assignment, e.g. `x += e`, which applies a binary operator before assignment. Note, however, that global variables in a specification are constant; they are assigned a value at initialization time and never change thereafter. Within a function body, we can read global variables and the function arguments, but we may only modify declared function-local variables.

Arrays held in local variables can be modified by the statement `x[i] = e`, which constructs a new array value by setting the `i`'th value in the current value of the array `x` to the result of evaluating `e` and writes this new array value into `x`. We may also modify a slice of an array by writing `x[i:j] = e`, which copies the array value resulting from evaluating `e` into the indicated slice of array `x`.

Conditional expressions and for-loops are standard, except that we allow the use of `elif` to combine an `else` with a subsequent `if`, and we allow the use of `break` to escape from a loop early. Each branch of a conditional, and each loop body is itself a sequence of statements, one on each line and indented appropriately.

Functions are defined with types annotating their arguments and results: `def f(x1:t1,...,xn:tn)→t` declares a new function called `f` with `n` arguments `x1,...,xn` with types `t1,...,tn`, and returns a result of type `t`. The body of the function is a sequence of statements; notably, the statement `return e` returns from the function with the evaluation of `e` as the result.

Each specification has a name (the name of the file in which it resides) and a sequence of statements that defines a cryptographic algorithm. Specifications can refer to other specifications by importing selected variables from the other specification into its own namespace by writing e.g. `from otherspec import function1`.

3.2 Example: Poly1305

Using the syntax defined above we can now write specifications like the Poly1305 MAC algorithm [1]. An excerpt of this hacspec, showing the core polynomial computation, appears in Spec 1.

The first 5 lines define the field of integers modulo the prime $\texttt{p} = 2^{130} - 5$. Line 1 declares and initializes the global variable `p`. The functions `fadd` (Lines 2-3) and `fmul` (Lines 4-5) define addition and multiplication in the field (modulo `p`). Both functions take two arguments of type `int` and return an `int`.

The function `poly` evaluates a polynomial over the prime field at a particular field element and returns the result. A polynomial of degree n is represented by an array of integers, where the 0'th element of the array contains the n'th coefficient, the $(n-1)$'th element contains the first coefficient, and the coefficient with degree 0 is assumed to be 0.

In the Poly1305 MAC algorithm, this array of coefficients is an encoding of the plaintext (`text`), and the value at which it is being evaluated is the MAC key (`key`). The body of the `poly` function first declares the local variable `result` of type `int` and initializes it to 0. It then iterates over the length of `text` with

Spec 1. Poly1305 hacspec

```
1  p:int = 2 ** 130 - 5
2  def fadd(x:int, y:int) -> int:
3      return (x + y) % p
4  def fmul(x:int, y:int) -> int:
5      return (x * y) % p
6
7  def poly(text:vlarray_t(int), key:int) -> int:
8      result: int = 0
9      for i in range(array.length(text)):
10         result = fmul(key,fadd(result, text[i]))
11     return result
```

a for-loop. The loop body (Line 10) updates the value of `result` by multiplying (`fmul`) the key with the sum (`fadd`) of the previous `result` and the i-th coefficient in `text`. Thus, for a `text` of size n, this function computes the field element:

$$\texttt{key}^n \cdot \texttt{text}[0] + \texttt{key}^{n-1} \cdot \texttt{text}[1] + \ldots + \texttt{key} \cdot \texttt{text}[n-1] \texttt{ modulo p}.$$

3.3 The Builtin Library

hacspec comes with a builtin library that implements many common functionalities needed by cryptographic algorithms. The full list of provided functions and types can be found in the hacspec documentation.[1] While it is possible to write specs entirely with the syntax defined in Table 1, the builtin library makes writing precise specs easier and simplifies compilation and formal proofs.

The set of cryptographic primitives (specs) implemented while working on hacspec (see Section 6 for details on the currently implemented primitives) can be used as well but are not provided as a library right now.

Modular Arithmetic. As we saw in Spec 1, cryptographic algorithms often rely on arithmetic modulo some (large, not necessarily prime) integer. The builtin library defines a type called `natmod_t(n)`, representing natural numbers modulo `n` (the set $\{0, \ldots, n-1\}$). It also defines modular arithmetic operations `+`, `-`, `*`, `**` (addition, subtraction, multiplication, exponentiation) on values of this type.

Hence, using the `natmod_t` type, we can rewrite the Poly1305 specification as shown in Spec 2. We first define a type `felem_t` for the field elements (naturals modulo `p`). We can then directly use `*` and `+` for field operations, without needing to define `fadd` and `fmul`. Furthermore, the spec is now more precise: it requires that the arguments of `poly` are in the prime field and guarantees that the result of polynomial evaluation is a field element, and violations of this post-condition can and should be checked in the spec and its implementations.

[1] `https://hacs-workshop.github.io/hacspec/docs/`

Spec 2. Poly1305 hacspec using the builtin library

```
p = nat((2 ** 130) - 5)
felem_t = natmod_t(p)
def poly(text:vlarray_t(felem_t), key:felem_t) -> felem_t:
    result = natmod(0,p)
    for i in range(array.length(text)):
        result = key * (result + text[i])
    return result
```

Machine Integers. A special case of `natmod_t(n)` are integers of a certain bit-length, i.e. when `n` is of the form 2^l. In particular, many cryptographic algorithms are defined over machine integers like unsigned 32-bit integers, which correspond to `natmod_t(2 ** 32)`. The builtin library defines types for commonly used machine integers like `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, and `uint128_t` as well as an arbitrary-length unsigned integer type `uintn_t(n)`. These integers provide the usual arithmetic operators like `natmod_t` but further provide bitwise operators `^, |, &, <<, >>` (xor, or, and, left-shift, right-shift) and the unary operator $\sim$ (bitwise negation). In addition, the library defines functions for rotating machine integers, and converting between integers of different sizes.

Byte-arrays, Vectors, and Matrices. The library defines abbreviations for several commonly-used variants of arrays. For example, the type `array_t(t,l)` represents arrays of type `t` and length `l`, a special case of the `vlarray_t(t)` type. The type `vlbytes_t` represents variable-length byte-arrays (`vlarray_t(uint8_t)`), which are commonly used as inputs and outputs of cryptographic algorithms. The corresponding fixed-length byte-array type is `bytes_t(l)`. These byte-array types provide library functions that convert byte-arrays to and from arrays of machine integer values; for example, the functions `bytes.from_uint32s_le` and `bytes.to_uint32s_le` inter-convert byte-arrays and `uint32_t` arrays, interpreted in little-endian order.

Another useful special case of arrays are vectors (`vector_t(t,l)`), which represent fixed-length arrays of numeric types. The content of a vector is either an integer (`int`), or `natmod_t(n)`, or `uintn_t(l)`, or another vector type. The advantage of using vectors is that they offer pointwise arithmetic operators (`+,-,*,**`), as well as standard functions like dot-products. We can build multi-dimensional vectors as vectors of vectors, but the builtin library provides a special type `matrix_t(t,rows,cols)` for two-dimensional vectors, for which it defines matrix-vector and matrix-matrix multiplication functions.

Type Abbreviations, Refinements, and Contracts. For the most part, specification authors can use just the types in the standard library to write their specs and use local type abbreviations when needed, for clarity, like the `felem_t` type in Spec 2. However, in many cases we may want to define a more specific type that represents a subset or a *refinement* of an existing type. The builtin library already provides commonly used refinements, like `nat_t` (integers ≥ 0), `pos_t` (integers > 0) and `range_t(s,e)` (integers from `s` to `e` $- 1$). In addition, it provides a construct for users to define their own refinement types.

The type `refine_t(t,f)` takes a type `t` and a predicate `f` from type `t` to `bool`, and returns a new type that represents values of type `t` that satisfy `f`. Elements of this new type can be implicitly converted to elements of type `t` but not vice-versa. Refinement types can be used to encode arbitrary logical invariants and are extensively used in verification frameworks like F* and Coq. When a refinement type is used on a function argument, it becomes a pre-condition for calling the function; when it is used in the result type, it becomes a post-condition guaranteed by the function. Since refinement predicates can be arbitrarily complex, checking refinement types is not always easy. We show how these types can be checked at run-time, using the python plugin Typeguard, or can be verified statically with powerful verification tools like F*.

Not all function specifications can be written using refinements. For example, refinements cannot specify how the result of a function relates to its arguments. We are experimenting with a general contract mechanism for hacspec, implemented as annotations, that can fill this gap.

4 Checking and Executing hacspec

We expect the hacspec language to be used at two levels. Standards authors and cryptographic researchers may write a hacspec for a new cryptographic algorithm and include it as pseudocode in a standard or in a research paper. They would like to easily check their spec for syntax and basic type errors and test it for correctness errors. For such users we provide tools written in python so that they do not need to install or learn any new tool to be able to achieve their goals.

More advanced users will want to check their hacspec for deeper properties, ranging from logical invariants to cryptographic security. They may also want to verify that an implementation of the algorithm, written in C or assembly, is correct with respect to the hacspec. For these users, we provide a syntax and static type checker, and compilers to various verification frameworks. To run these tools, one must install OCaml as well as the target verification tool. Note that verifying existing implementations requires a compilation target such as cryptol or VST that do not exist yet.

In this section, we describe our tool-set for checking and running hacspec and highlight the main challenges in implementing these tools.

4.1 Syntax and Static Type Checking

Since hacspec is a subset of python, specification authors used to python may be tempted to use python features that are not supported by our domain-specific language. To check that a hacspec is valid we have built a python tool called *hacspec-check* that is part of the hacspec python package. This checker enforces the syntax in Table 1; it finds and points out invalid expressions, types, and statements, and forbids the use of external values and functions that are not included in the builtin library.

The syntax checker only ensures that the type annotations in a hacspec have the right syntax, it does not check that the code is type-correct. We wrote an additional tool in OCaml that statically enforces the hacspec type system. It requires that all variables be declared before use, and that they are used in accordance with their declared type. This tool does not perform any inference, but still, it must account for basic types (like `int`), builtin types (like `uint32_t`), and parameterized types (like `array_t(t,l)`). The checker also knows and enforces the types for all the builtin operators and functions in the builtin library. In particular, the type-checker prevents aliasing of mutable data structures like arrays, by ensuring that two array variables never point to the same array value — when assigning an array variable, the assigned value must be a fresh copy of an array.

4.2 Executing hacspec with Run-time Checks

Because hacspec uses python syntax it can be directly executed using the python runtime. We provide a python implementation, called `speclib.py`, for the builtin library. To make hacspec easy to use for spec authors the library and the `hacspec-check` command line tool are bundled in the hacspec python package.[1] After installing this package, the spec author can include the standard library: `from hacspec.speclib import *` and write and run specifications.

Our python library uses python classes to implement the various types and type constructors: `natmod_t`, `uintn_t`, `array_t`, etc. Each class defines the relevant unary and binary operators (e.g. `+`) for a type by overloading the corresponding python "magic" methods (e.g. `__add__`) in that class. We define each of these operator methods to first check that the left and right expressions are both of the expected type, before calculating the result and casting it to the expected return type. Python also allows us to define custom getters and setters for arrays using the standard syntax (e.g. `a[1] += b[1]`), and we overload these methods to check that the array and the assigned values have a consistent type, and that all indexes are within bounds, before executing the expected array access operation. Vectors and matrices are sub-classes of arrays but are provided with point-wise binary operators that are subject to further checks.

Although python allows a rich type annotation syntax (allowing arbitrary expressions to compute types), it does not itself check these types. However, there are various tools and plugins that can check type annotations statically or at runtime. For example, mypy[2] can perform basic static type checking, but unfortunately it does not not support everything needed by hacspec, e.g., generic types, custom types, or type aliases. Instead, we use run-time type checking with typeguard[3], which allows spec authors to quickly get feedback on the correctness of their used types without having to use an additional execution environment.

Typeguard checks functions that are annotated with the `@typechecked` decorator for conformance to their declared types. We annotate all the functions in

[1] `https://pypi.org/project/hacspec/`
[2] `http://www.mypy-lang.org/`
[3] `https://github.com/agronholm/typeguard/`

our builtin library and hence require that they be used correctly. The same is required from all user-written hacspec functions. In addition to the checks built into Typeguard, our library functions themselves perform a variety of builtin semantic checks that can detect common coding errors in hacspec specifications. For example; Every array access is bound-checked to make sure that it is not over- or under-running the array; When modifying array elements type checks ensure that only compatible values are added; Arithmetic operations ensure that only operands of appropriate bit-lengths and moduli are used, e.g., rotate and shift functions ensure that the shift values are positive and not greater than the integers bit-length; Using refinement types ensures that the type is correct and within the refined value range.

For each specification we encourage authors to provide an extensive test-suite as a list of positive and negative test vectors. These should include all the test-vectors provided in the standard but also other test-suites, such as those provided by NIST[4] or open-source projects like Wycheproof[5]. Running tests on the code with run-time type-checking can make the execution very slow but provides higher assurance in the spec. Our specs can also be tested in an optimized mode that disables these run-time type-checks.

5 Verifying hacspec

For advanced users, we describe two verification tools we are currently building for hacspec: one for verifying implementations using F* and the other for building cryptographic proofs in EasyCrypt. Both tools are based on the typed AST, generated by the hacspec static type-checker.

5.1 F* Compiler

F* is a general-purpose programming language targeted at building verified software [23]. In particular, it has been used to build a verified cryptographic library called HACL* [25]. For each cryptographic algorithm HACL* includes an optimized stateful implementation written in a subset of F* called Low*. Code written in this subset can be compiled to C code and hence used within cryptographic protocol implementations, such as TLS libraries.

The Low* implementation of each algorithm is verified for functional correctness against a formal specification written in F*. Unlike the implementation, the specification is pure and total; i.e. it cannot have any side-effects and must always terminate. In addition to correctness, Low* code is also proved to be memory safe and have secret independent execution time, i.e. it does not branch on secret values or access memory at secret addresses.

HACL* includes hand-written specifications for all the algorithms in the library. Our goal is to replace these specifications with those written in hacspec

[4] `https://csrc.nist.gov/Projects/Cryptographic-Algorithm-Validation-Program`

[5] `https://github.com/google/wycheproof`

and to make it easier for new HACL⋆ specifications to be compiled from hacspec. To this end we develop a compiler from hacspec to F⋆ specifications.

First, we implement the hacspec builtin library as an F⋆ module (`SpecLib.fst`), so that the compiled F⋆ specifications can also have access to all the types and functions used in the source specification. This library module defines hacspec types like `int`, `natmod_t`, `vlarray_t` etc. in terms of the corresponding types in the F⋆ standard library. In particular, hacspec arrays (`vlarray_t(t)`) are implemented as immutable sequences (`seq t`) in F⋆. We implement all unary and binary operators using function overloading in F⋆.

Then, we implement a translation from hacspec specifications to F⋆ modules that syntactically transforms each value, expression, and statement to the corresponding syntax in F⋆. For example, the Poly1305 specification in Spec 2 translates to the F⋆ specification in Spec 3. The main syntactic change in the F⋆ version is that all statements that modify local variables get translated to pure expressions (in a state-passing style) that redefine the local variables (using scoped let-expressions) instead of modifying them. For loops are translated to an application of the higher-order `repeati` combinator, which applies a function a given number of times to the input.

Spec 3. Poly1305 hacspec using the builtin library

```
let p : nat = (2 ** 130) - 5
type felem_t = natmod_t(p)
let poly (text:vlybtes_t) (key:felem_t) : felem_t =
    let result = natmod(0, p) in
    repeati (length text)
            (fun i result -> key * (result + text.[i]))
            result
```

In addition to the syntactic translation of the code, the F⋆ compiler translates hacspec refinement types to refinement types in F⋆, so that the F⋆ typechecker can verify all of them statically using an SMT solver.

In a typical workflow the standard author writes a hacspec for a cryptographic algorithm, translates it to F⋆, and typechecks the result with F⋆ to find progamming errors. The crypto developer then writes an optimized implementation of the algorithm in Low⋆ and verifies that it is memory-safe, secret independent, and that it conforms to the specification derived from the hacspec. For example, a Low⋆ implementation of Poly1305 would first need to implement the modular arithmetic in `natmod_t` with prime-specific optimized bignum arithemetic. The developer would then typically write a stateful version of `poly` that modified the `result` in-place. Proving that this implementation matches the specification is a challenging task, but one made easier by libraries of verified code in HACL⋆.

5.2 EasyCrypt Compiler

EasyCrypt is a proof assistant for verifying the security of cryptographic constructions in the computational model. EasyCrypt adopts the code-based approach [5], in which security goals and hardness assumptions are modeled as

probabilistic programs (called experiments or games) with unspecified adversarial code. EasyCrypt uses formal tools from program verification and programming language theory to rigorously justify cryptographic reasoning.

EasyCrypt is composed of several ingredients: i) a simply-typed, higher-order, pure functional language that forms the logical basis of the framework; ii) a probabilistic While language that allows the algorithmic description of the schemes under scrutiny; and iii) programming language logic. These logic include a probabilistic, relational Hoare logic, relating pairs of procedures; a probabilistic Hoare logic allowing one to carry out proofs about the probability of some event during the execution of a procedures; and an ordinary (possibilistic) Hoare logic.

The compiler from hacspec to EasyCrypt is composed of two phases. First, the types and procedures are translated from the hacspec syntax to the relevant EasyCrypt constructions. Since EasyCrypt enjoys a simply-typed language discipline, nearly all refinements have to be pruned during this phase. However, we have some special support for types that depend on fixed values: in that case, we define a generic theory that encloses the dependent type and use the EasyCrypt theory cloning for generating instances of that type for some fixed values. This translation relies on a EasyCrypt library that mirrors `speclib`. When the translator detects that a procedure is pure, deterministic and terminates (w.r.t. the EasyCrypt termination checker), this one is translated directly as a logical operator. In all other cases, hacspec procedures are translated to While-based EasyCrypt ones. Some constructions ares not supported by the current compiler. For example, EasyCrypt does not allow the definition of recursive procedures and inner procedures and we currently simply reject them. We plan to modify the compiler so that it encodes these constructs as valid EasyCrypt programs.

In a second phase, top-level refinements (i.e. refinements that apply to the arguments or result types of a procedure) are translated into EasyCrypt statements using the probabilistic Hoare logic. We give, in Spec 4, the Poly1305 specification in Spec 2 translated to EasyCrypt.

Spec 4. Poly1305 hacspec using the builtin library

```
1  op p : int = 2 ^ 130 - 5.
2
3  clone import NatMod as FElem_p with op size <- p.
4
5  module Spec = {
6    proc poly(text : byte array, key : felem_t) : felem_t = {
7      var result = FElem.zero, i;
8      i = 0; while (i < Array.length text) {
9        result = key * (result + FElem.mk (Byte.to_int (text.[i])));
10       i = i + 1;
11     }
12     return result;
13   }
14 }.
```

From that point, it is then possible to use EasyCrypt to prove these statements, hence verifying the soundness of the hacspec refinements directly in EasyCrypt. However, we expect developers to follow a two-tier workflow. First, developers can use the F⋆ compiler of Section 5.1 and derive the correctness of the refinements using the F⋆ type-checker. Because both F⋆ and EasyCrypt work on the same specification by construction (they are obtained from the compilation of the same hacspec procedure), this step enforces that the refinements that are translated as Hoare statement into EasyCrypt are sound. In a second step, developers could take advantage of EasyCrypt to prove the security of the obtained primitives. Indeed, EasyCrypt comes with all the necessary materials (relational probabilistic logic, libraries of standard cryptographic games, experiments & arguments) for the study of cryptographic primitives in the computational model, using the now standard game-hopping technique. For example, in addition to Poly1305, one could also write in hacspec the Chacha20 encryption scheme, obtain their EasyCrypt counter-part, and prove in the same system that they provide a secure AEAD (Authenticated Encryption with Associated Data) scheme. This, along with the verified Low⋆ implementation of Chacha20-Poly1350, would lead to an efficient and formally proven secure AEAD scheme.

Note that by the different nature of the two involved languages (hacspec & EasyCrypt), the obtained EasyCrypt specification might not be in total adequacy with EasyCrypt idioms. However, using EasyCrypt relational logic, one can start by first proving that the hacspec specifications simulate a hand-written, more natural, EasyCrypt one. Although this requires more work to link the EasyCrypt proofs to the hacspec specifications, it gives a formal link between the two.

6 Evaluation, Conclusion, and Future Work

To evaluate hacspec we implemented several standardized cryptographic algorithms using only the builtin library. Table 2 summarizes the specifications we currently have written and tested in hacspec as well as their F⋆ version if available. Even more complex algorithms such as Kyber or the SHA2 and SHA3 function families (with all their variants including SHAKE), as well as combined algorithms like the AES-GCM and ChaCha20-Poly1305 AEAD can be written in less than 300 lines of hacspec. (The combined AEAD algorithms in Table 2 list the lines of code for the combination of cipher and MAC as well as the combined lines of code of cipher and MAC.) Modern algorithms that are designed to have rather straight-forward implementations like Poly1305 or Curve25519/Curve448 can be implemented in 50 to 70 lines of code. This shows that hacspec allows concise specifications of common cryptographic algorithms.

The last column in Table 2 lists the lines of F⋆ code that is produced when compiling the hacspec code. Not all specs can be compiled to F⋆ at this point because some hacspec constructs are not fully supported by the compiler yet. The hash algorithms SHA2, SHA3, and Blake2 for example use inner functions for

concise handling of multiple digest lengths, which can not be properly compiled to F⋆ at the moment.

hacspec	Category	Standard	LoC	F⋆ LoC
speclib	Library	-	849	213
AES	Symmetric Cipher	NIST FIPS 197	190	210
GCM	MAC	NIST SP 800-38D	61	46
AES-GCM 128	AEAD	NIST SP 800-38D	47 + 251	-
Chacha20	Symmetric Cipher	IETF RFC 7539	87	89
Poly1305	MAC	IETF RFC 7539	43	37
Chacha20-Poly1305	AEAD	IETF RFC 7539	45 + 130	-
SHA-2	Hash	NIST FIPS 180-2	192	-
SHA-3	Hash	NIST FIPS 202	193	-
Blake2	Hash	IETF RFC 7693	162	-
Curve25519	ECDH	IETF RFC 7748	87	100
Curve448	ECDH	IETF RFC 7748	69	63
P256	ECDH/Signature	NIST SP 800-56A	102	-
Ed25519	Signature	IETF RFC 8032	182	-
RSA-PSS	Signature	IETF RFC 8017	151	-
Kyber	Post-Quantum KEM	NIST PQ Challenge	285	-
Frodo	Post-Quantum KEM	NIST PQ Challenge	192	-
WOTS+	One-time Signature	IETF RFC 8391	134	-

Table 2. Specifications written in hacspec

Conclusion. In this paper we described the goals and architecture of hacspec, a new specification language for cryptographic algorithms. We defined hacspec as a domain-specific language with minimal syntax that can be interpreted by the python runtime. We showed that together with a builtin library hacspec allows to write succinct specifications of cryptographic algorithms. To allow spec authors to use hacspec without much effort we provide an implementation of the builtin library in python and a tool to check the spec syntax. To verify specs written in hacspec we showed how to compile specs to F⋆ and EasyCrypt. This enables formal proofs of correctness of code with respect to the spec, proves of protection against side channels and memory issues, as well as cryptographic security proves.

Future Work. The compiler and builtin library are still in early stages and are likely to evolve over time. Additional library features such as function contracts are in development. The goal of this work is to describe the current state of hacspec and invite spec authors to use it and give feedback to guide future development of the hacspec language, the builtin library, and tooling. We would

like to see more compilers to other formal languages such as cryptol and Coq to allow formal proofs of specifications using hacspec in those frameworks.

Acknowledgments. We would like to thank all participants of the HACS workshop that made this work possible. hacspec is an ongoing project with a number contributors in addition to the the authors.

Online Materials.
hacspec source code: `https://github.com/hacs-workshop/hacspec/`
hacspec builtin library documentation: `https://hacs-workshop.github.io/hacspec/docs/`
hacspec mailing list: `https://moderncrypto.org/mailman/listinfo/hacspec`

References

1. ChaCha20 and Poly1305 for IETF Protocols. IETF RFC 7539 (2015)
2. Elliptic Curves for Security. IETF RFC 7748 (2016)
3. Almeida, J., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B.: Jasmin: High-assurance and high-speed cryptography. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS. p. 1807–1823. to appear (2017), `https://acmccs.github.io/papers/p1807-almeidaA.pdf`
4. Appel, A.W.: Verified software toolchain. In: NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings. p. 2 (2012)
5. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.: Easycrypt: A tutorial. In: Aldini, A., López, J., Martinelli, F. (eds.) Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures. Lecture Notes in Computer Science, vol. 8604, pp. 146–166. Springer (2013). https://doi.org/10.1007/978-3-319-10082-1_6, `https://doi.org/10.1007/978-3-319-10082-1_6`
6. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.Y.: EasyCrypt: A Tutorial, pp. 146–166. Springer International Publishing, Cham (2014)
7. Bernstein, D.J.: Cache-timing attacks on aes. Tech. rep. (2005)
8. Blanchet, B.: A computationally sound mechanized prover for security protocols. IEEE Transactions on Dependable and Secure Computing **5**, 193–207 (06 2007)
9. Böck, H., Zauner, A., Devlin, S., Somorovsky, J., Jovanovic, P.: Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS. In: 10th USENIX Workshop on Offensive Technologies (WOOT 16). USENIX Association, Austin, TX (2016), `https://www.usenix.org/conference/woot16/workshop-program/presentation/bock`
10. Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K.R.M., Lorch, J.R., Parno, B., Rane, A., Setty, S., Thompson, L.: Vale: Verifying high-performance cryptographic assembly code. In: Proceedings of the USENIX Security Symposium (Aug 2017)
11. US Department of Commerce, N.I.o.S., (NIST), T.: Federal Information Processing Standards Publication 180-4: Secure hash standard (SHS) (2012)
12. Courtois, N.T., Emirdag, P., Valsorda, F.: Private key recovery combination attacks: On extreme fragility of popular bitcoin key management, wallet and cold storage solutions in presence of poor rng events. Cryptology ePrint Archive, Report 2014/848 (2014), `https://eprint.iacr.org/2014/848`

13. Dworkin, M.: Recommendation for Block Cipher Modes of Operation: Galois/-Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D (2007)
14. Dworkin, M.J., Barker, E.B., Nechvatal, J.R., Foti, J., Bassham, L.E., Roback, E., Jr., J.F.D.: Advanced Encryption Standard (AES). NIST FIPS-197 (2001)
15. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic – with proofs, without compromises. In: S&P'19: Proceedings of the IEEE Symposium on Security & Privacy 2019 (May 2019), http://adam.chlipala.net/papers/FiatCryptoSP19/
16. Institute, A.N.S.: Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm. ANSI X9.62-1998 (199)
17. Josefsson, S., Liusvaara, I.: Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032 (Informational) (Jan 2017). https://doi.org/10.17487/RFC8032, https://www.rfc-editor.org/rfc/rfc8032.txt
18. Käsper, E., Schwabe, P.: Faster and timing-attack resistant aes-gcm. In: Clavier, C., Gaj, K. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2009. pp. 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
19. Langley, A., Hamburg, M., Turner, S.: Elliptic Curves for Security. RFC 7748 (Informational) (Jan 2016). https://doi.org/10.17487/RFC7748, https://www.rfc-editor.org/rfc/rfc7748.txt
20. Mouha, N., Raunak, M.S., Kuhn, D.R., Kacker, R.: Finding bugs in cryptographic hash function implementations. Cryptology ePrint Archive, Report 2017/891 (2017), https://eprint.iacr.org/2017/891
21. Petcher, A., Morrisett, G.: The foundational cryptography framework. In: Principles of Security and Trust. pp. 53–72. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
22. Petcher, A., Morrisett, G.: The foundational cryptography framework. In: Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings. pp. 53–72 (2015)
23. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 256–270
24. Tomb, A.: Automated verification of real-world cryptographic implementations. IEEE Security and Privacy **14**(6), 26–33 (2016)
25. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: A verified modern cryptographic library. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1789–1806 (2017)