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

Denis Merigoux[1] Franziskus Kiefer[2] Karthikeyan Bhargavan[1]

Inria Paris[1], Wire[2]

April 12th, 2021

A tale of two worlds

Rust is memory safe

Let's write high-assurance cryptography in Rust!

A tale of two worlds

Rust is memory safe

Let's write high-assurance cryptography in Rust!

The shiny new Rust band

- RustCrypto
- dalek-cryptography
- ring
- rustpq
- RusTLS

The old verified guard [3]

- Evercrypt/HACL*/Vale [10, 8]
- Fiat-crypto [5]
- JasminCrypt [2]

A tale of two worlds

Rust is memory safe

Let's write high-assurance cryptography in Rust!

The shiny new Rust band

- RustCrypto
- dalek-cryptography
- ring
- rustpq
- RusTLS

The old verified guard [3]

- Evercrypt/HACL*/Vale [10, 8]
- Fiat-crypto [5]
- JasminCrypt [2]

How to connect both worlds ?

Right now: the specification problem

From verified implementations to Rust

Simply provide Rust bindings (e.g. `https://crates.io/crates/evercrypt`)

Right now: the specification problem

From verified implementations to Rust

Simply provide Rust bindings (e.g. `https://crates.io/crates/evercrypt`)

Verified Rust implementations?

Rust verification tools still maturing, hopefully one day they handle full functional correctness for crypto libraries with 10-100k lines of code.

Right now: the specification problem

From verified implementations to Rust

Simply provide Rust bindings (e.g. `https://crates.io/crates/evercrypt`)

Verified Rust implementations?

Rust verification tools still maturing, hopefully one day they handle full functional correctness for crypto libraries with 10-100k lines of code.

Functional correctness specifications

Achilles' heel of verified implementations: specifications. Usually written in pseudocode, ambiguous. Attempt to convert to Python but little traction [4] (because of Python?).

Bringing the two worlds together

Idea/Hypothesis

Cryptographic code is DSL-friendly (Low* [9], Jasmin [1], Usuba [6])
Let's make an embedded Rust DSL!

Bringing the two worlds together

Idea/Hypothesis

Cryptographic code is DSL-friendly (Low* [9], Jasmin [1], Usuba [6])
Let's make an embedded Rust DSL!

For cryptographers

- Convenient tooling to write executable specifications and/or reference implementations
- Effortless switch to optimized native Rust implementations.

For proof people

- Specifications reviewed by domain experts.
- Reduced Trusted Computing Base for proof developments

A taste of hacspec

```
fn chacha_line(
  a: StateIdx,
  b: StateIdx,
  d: StateIdx,
  s: usize,
  m: State
) -> State {
  let mut state = m;
  state[a] = state[a] + state[b];
  state[d] = state[d] ^ state[a];
  state[d] =
    state[d].rotate_left(s);
  state
}
```

A taste of hacspec

```
fn chacha_line(
  a: StateIdx,
  b: StateIdx,
  d: StateIdx,
  s: usize,
  m: State
) -> State {
  let mut state = m;
  state[a] = state[a] + state[b];
  state[d] = state[d] ^ state[a];
  state[d] =
    state[d].rotate_left(s);
  state
}
```

```
pub fn poly(m: &ByteSeq, key: KeyPoly) -> Tag {
  let r = le_bytes_to_num(
    &key.slice(0, BLOCKSIZE));
  let r = clamp(r);
  let s = le_bytes_to_num(
    &key.slice(BLOCKSIZE, BLOCKSIZE));
  let s = FieldElement::from_secret_literal(s);
  let mut a = FieldElement::from_literal(0u128);
  for i in 0..m.num_chunks(BLOCKSIZE) {
    let (len, block) =
      m.get_chunk(BLOCKSIZE, i);
    let block_uint = le_bytes_to_num(&block);
    let n = encode(block_uint, len);
    a = a + n;
    a = r * a;
  }
  poly_finish(a, s)
}
```

The hacspec DSL – https://hal.inria.fr/hal-03176482 [7]

$$
\begin{array}{lcl}
p & ::= & [i]^* \\
i & ::= & \texttt{array!}(\ t\ ,\ \mu\ ,\ n \in \mathbb{N}\ ) \\
 & | & \texttt{fn}\ f\ (\ [d]^+\ )\ \texttt{->}\ \mu\ b \\
d & ::= & x\ :\ \tau \\
\mu & ::= & \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \\
 & | & \texttt{Seq}<\ \mu\ > \\
 & | & t \\
 & | & (\ [\mu]^+\ ) \\
\tau & ::= & \mu \\
 & | & \&\mu
\end{array}
$$

The hacspec DSL – https://hal.inria.fr/hal-03176482 [7]

$$
\begin{array}{lcl}
p & ::= & [i]^* \\
i & ::= & \texttt{array!}(\, t\, ,\ \mu\, ,\ n \in \mathbb{N}\,) \\
 & | & \texttt{fn}\ f\,(\,[d]^+\,)\ \texttt{->}\ \mu\ b \\
d & ::= & x : \tau \\
\mu & ::= & \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \\
 & | & \texttt{Seq}\texttt{<}\ \mu\ \texttt{>} \\
 & | & t \\
 & | & (\,[\mu]^+\,) \\
\tau & ::= & \mu \\
 & | & \&\mu \\
b & ::= & \{\ [s\,;]^+\ \} \\
s & ::= & \texttt{let}\ x : \tau = e \\
 & | & x = e \\
 & | & \texttt{if}\ e\ \texttt{then}\ b\ (\,\texttt{else}\ b) \\
 & | & \texttt{for}\ x\ \texttt{in}\ e\ ..\ e\ b \\
 & | & x[\,e\,] = e \\
 & | & e \\
 & | & b
\end{array}
$$

The hacspec DSL – `https://hal.inria.fr/hal-03176482` [7]

$$
\begin{array}{rcl}
p & ::= & [i]^{*} \\
i & ::= & \texttt{array!}(\ t,\ \mu,\ n \in \mathbb{N}\ ) \\
 & \mid & \texttt{fn}\ f\ (\ [d]^{+}\ )\ \texttt{->}\ \mu\ b \\
d & ::= & x : \tau \\
\mu & ::= & \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \\
 & \mid & \texttt{Seq<}\ \mu\ \texttt{>} \\
 & \mid & t \\
 & \mid & (\ [\mu]^{+}\ ) \\
\tau & ::= & \mu \\
 & \mid & \&\mu \\
b & ::= & \{\ [s\,;]^{+}\ \} \\
s & ::= & \texttt{let}\ x : \tau = e \\
 & \mid & x = e \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ b\ (\ \texttt{else}\ b) \\
 & \mid & \texttt{for}\ x\ \texttt{in}\ e\ ..\ e\ b \\
 & \mid & x[\ e\ ] = e \\
 & \mid & e \\
 & \mid & b
\end{array}
$$

$$
\begin{array}{rcl}
e & ::= & () \mid \texttt{true} \mid \texttt{false} \\
 & \mid & n \in \mathbb{N} \\
 & \mid & x \\
 & \mid & f\ (\ [a]^{+}\ ) \\
 & \mid & e \odot e \\
 & \mid & \oslash\ e \\
 & \mid & (\ [e]^{+}\ ) \\
 & \mid & e\,.(n \in \mathbb{N}) \\
 & \mid & x[\ e\ ] \\
a & ::= & e \\
 & \mid & \&e \\
\odot & ::= & + \mid - \mid * \\
 & \mid & / \mid \&\& \mid || \\
 & \mid & == \mid != \mid > \mid < \\
\oslash & ::= & - \mid \sim
\end{array}
$$

Simple call-by-value semantics with variable context

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

Simple call-by-value semantics with variable context

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

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	$p \vdash f\ (\ v_1\ , \ldots ,\ v_n\ ) \Downarrow v$

Linear typing with Rust specificities

Typing context	Γ	$::=$	$\varnothing$
(unordered map)		$\mid$	$x : \tau, \Gamma$
		$\mid$	$f : (\ [\tau]^{+}\ ) \rightarrow \mu, \Gamma$
Type dictionary	Δ	$::=$	$\varnothing \mid t \rightarrow [\ \mu;\ n \in \mathbb{N}\ ], \Delta$

Linear typing with Rust specificities

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

$$
\begin{array}{lc}
\text{Context splitting} & \Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \\
\text{Implementing the Copy trait} & \Delta \vdash \tau : \texttt{Copy}
\end{array}
$$

Linear typing with Rust specificities

Typing context	Γ	$::=$	$\varnothing$
(unordered map)		$\mid$	$x : \tau, \Gamma$
		$\mid$	$f : ([\tau]^{+}) \to \mu, \Gamma$
Type dictionary	Δ	$::=$	$\varnothing \mid t \to [\ \mu;\ n \in \mathbb{N}\ ], \Delta$

Context splitting	$\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$
Implementing the `Copy` trait	$\Delta \vdash \tau : \texttt{Copy}$

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

Implementation: AST or MIR?

MIR

- – Very desugared
- – – Basic blocks

AST

- ++ Close to the source code
- \+ Structured control flow

Implementation: AST or MIR?

MIR

- − Very desugared
- −− Basic blocks
- \+ Traits and methods resolved
- \+ Lifetime resolution

AST

- ++ Close to the source code
- \+ Structured control flow
- − No trait or methods support
- − No borrow checking support

Implementation: AST or MIR?

MIR

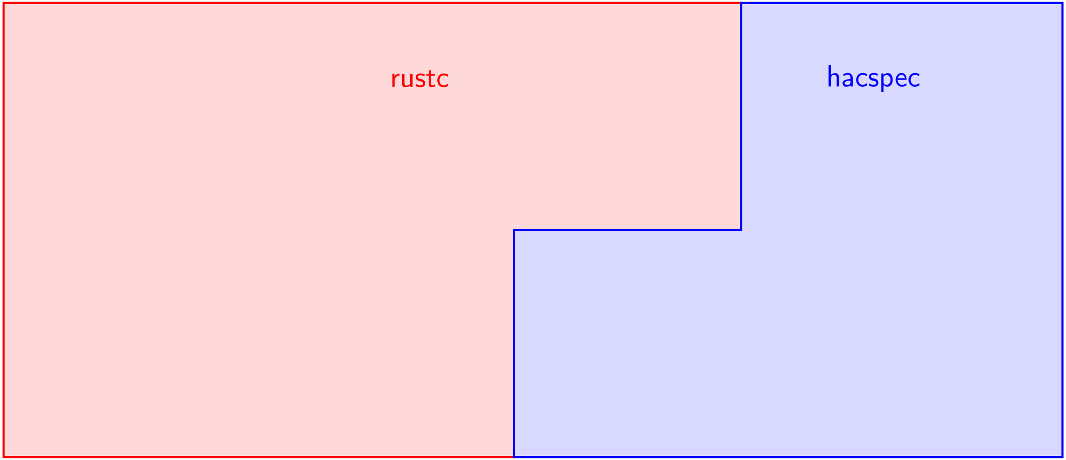
- − Very desugared
- −− Basic blocks
- + Traits and methods resolved
- + Lifetime resolution

AST

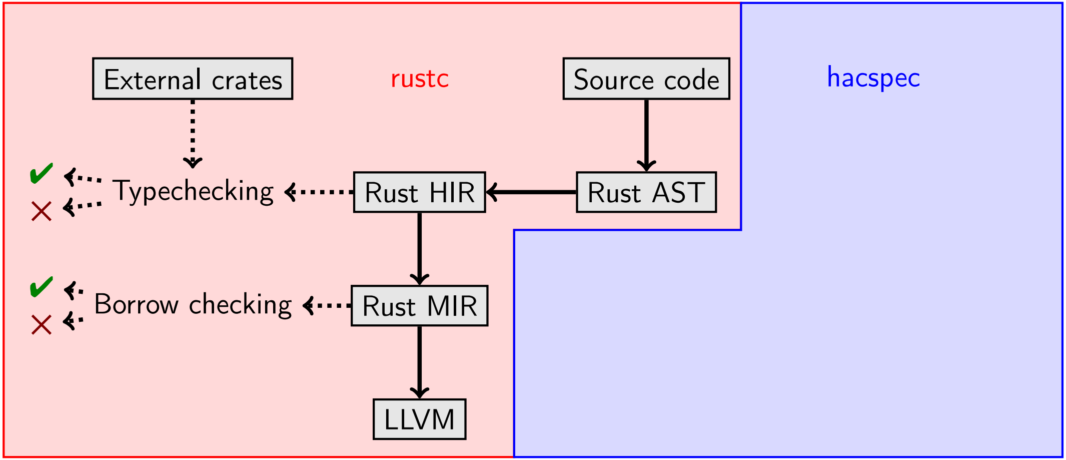
- ++ Close to the source code
- + Structured control flow
- − No trait or methods support
- − No borrow checking support

For originality (and our specific use), we choose AST!

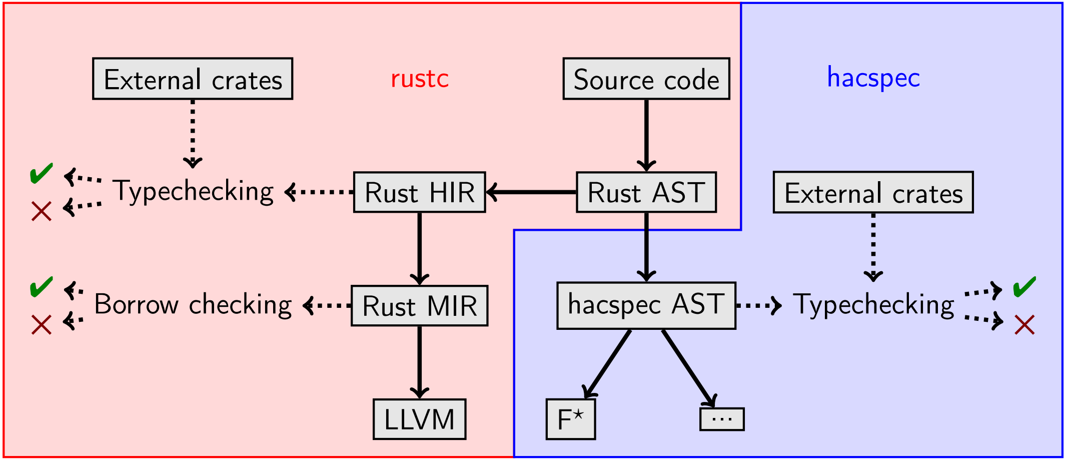
The hacspec typechecker



The hacspec typechecker



The hacspec typechecker



hacspec programs

Primitive / Lines of code (* with proofs)	hacspec	HACL*
ChaCha20	132	191
Poly1305	77	77
Chacha20Poly1305	59	89
NTRU-Prime	95	–
SHA3	173	227
SHA256	148	219
P256	172	370*
ECDSA-P256-SHA256	52	558*
Curve25519	107	124
HKDF	57	72
BLS-12-381	540	–
Gimli	241	–

Verification backend: F*

```
let chacha_line (a_4 : state_idx) (b_5 : state_idx)
  (d_6 : state_idx) (s_7 : uint_size{
    (**) s_7 > 0 && s_7 < 32
  }) (m_8 : state) : state =
let state_9 = m_8 in
let state_9 = array_upd state_9 (a_4) (
  (array_index (state_9) (a_4)) +. (array_index (state_9) (b_5)))
in
let state_9 = array_upd state_9 (d_6) (
  (array_index (state_9) (d_6)) ^. (array_index (state_9) (a_4)))
in
let state_9 = array_upd state_9 (d_6) (
  uint32_rotate_left (array_index (state_9) (d_6)) (s_7))
in
state_9
```

The hacspec libraries

secret-integers

- Wrapper around all signed and unsigned integers: `U8`, `I32`, etc.
- Forbids non-constant-time operations (parametricity)

The hacspec libraries

secret-integers

- Wrapper around all signed and unsigned integers: `U8`, `I32`, etc.
- Forbids non-constant-time operations (parametricity)

abstract-integers

- Verification-friendly bounded natural integers: `nat_mod!`
- Adapted for field-arithmetic-based specifications

The hacspec libraries

secret-integers

- Wrapper around all signed and unsigned integers: `U8`, `I32`, etc.
- Forbids non-constant-time operations (parametricity)

abstract-integers

- Verification-friendly bounded natural integers: `nat_mod!`
- Adapted for field-arithmetic-based specifications

hacspec-lib

- Copyable const-length arrays: `array!`
- Linear fixed-length arrays: `Seq`
- Traits and helpers for the hacspec writers, integrated with typechecker

Conclusion

Research collaboration

Inria (**Karthikeyan Bhargavan**, **Denis Merigoux**)
Wire (**Franziskus Kiefer**)
University of Porto (**Manuel Barbosa**)
Aarhus University (**Bas Spitters**)
MPI-SP (**Peter Schwabe**)

Objective

Bridging Rust cryptography with existing verification tools

Implementation philosophy

Embedded DSL capturing the functional part of Rust

Website
`hacspec.github.io`

Code
`github.com/hacspec/hacspec`

Technical report
`hal.inria.fr/hal-03176482`

Contact: `denis.merigoux@inria.fr`

References I

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub.
Jasmin: High-assurance and high-speed cryptography.
In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823, 2017.

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub.
The last mile: High-assurance and high-speed cryptographic implementations.
In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 965–982. IEEE, 2020.

References II

Manuel Barbosa, Gilles Barthe, Karthikeyan Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno.
Sok: Computer-aided cryptography.
In *IEEE Symposium on Security and Privacy (S&P'21)*, 2021.

Karthikeyan Bhargavan, Franziskus Kiefer, and Pierre-Yves Strub.
hacspec: Towards verifiable crypto standards.
In Cas Cremers and Anja Lehmann, editors, *Security Standardisation Research*, pages 1–20, Cham, 2018. Springer International Publishing.

Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala.
Simple high-level code for cryptographic arithmetic-with proofs, without compromises.
In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1202–1219. IEEE, 2019.

References III

Darius Mercadier and Pierre-Évariste Dagand.
Usuba: high-throughput and constant-time ciphers, by construction.
In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 157–173, 2019.

Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan.
Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust.
Technical report, Inria, March 2021.

Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, et al.
Evercrypt: A fast, verified, cross-platform cryptographic provider.
In *IEEE Symposium on Security and Privacy (SP)*, pages 634–653, 2020.

References IV

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, et al.
Verified low-level programming embedded in f.
Proceedings of the ACM on Programming Languages, 1(ICFP):1–29, 2017.

Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche.
Hacl*: A verified modern cryptographic library.
In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, 2017.