Modularity, Code Specialization, and Zero-Cost Abstractions for Program Verification

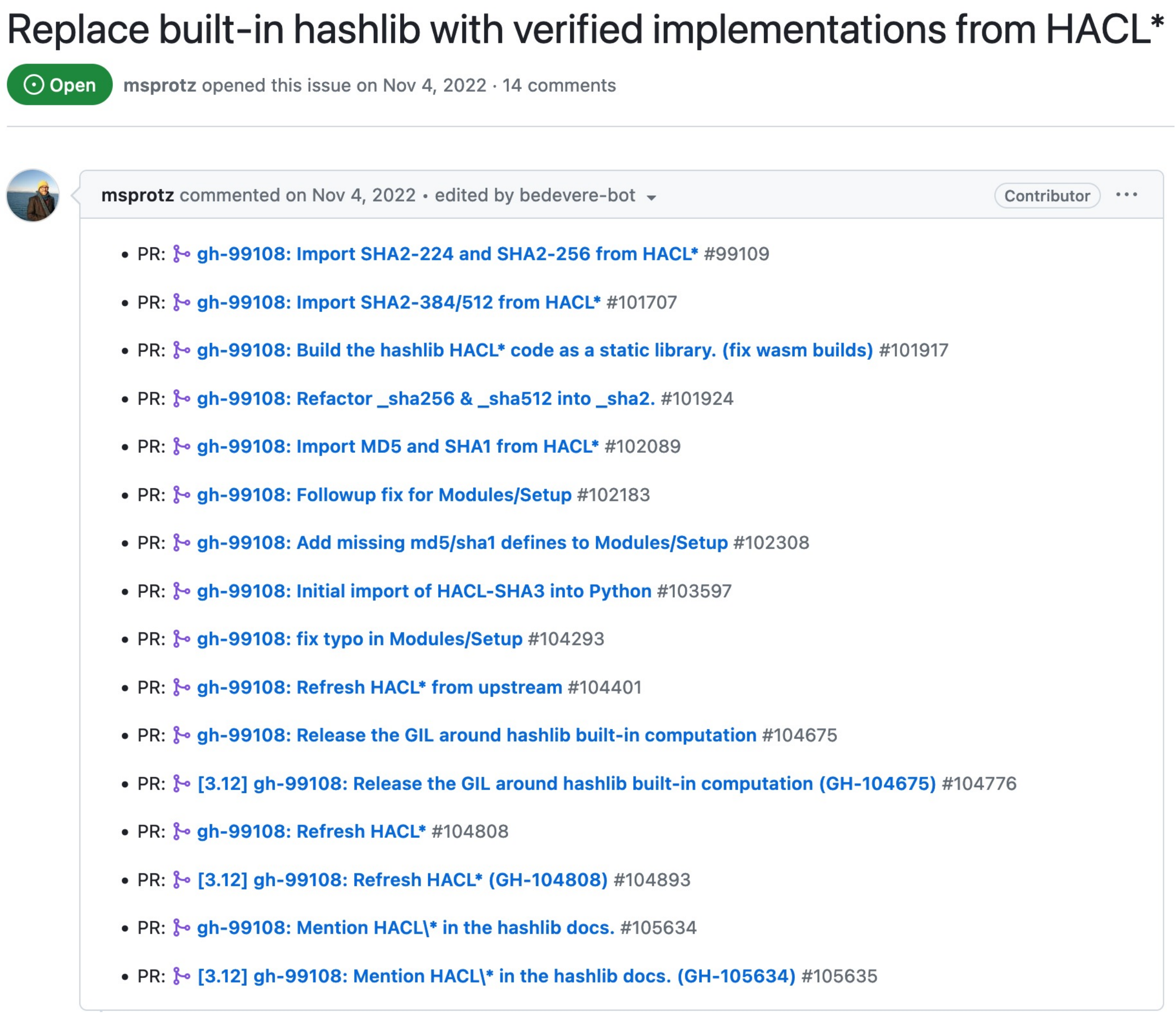
Son Ho (Inria)

Aymeric Fromherz (Inria)

Jonathan Protzenko (Microsoft Research)

Verified Crypto Coming to a Python near you

- Python 3.12
- This work: the story of how we built a layer of high-level APIs
- Technical ingredients: elaborator reflection, meta-programming, automated code rewriting, and high-level abstractions



Replace built-in hashlib with verified implementations from HACL*

Open msprotz opened this issue on Nov 4, 2022 · 14 comments

msprotz commented on Nov 4, 2022 • edited by bedevere-bot Contributor

- PR: gh-99108: Import SHA2-224 and SHA2-256 from HACL* #99109
- PR: gh-99108: Import SHA2-384/512 from HACL* #101707
- PR: gh-99108: Build the hashlib HACL* code as a static library. (fix wasm builds) #101917
- PR: gh-99108: Refactor _sha256 & _sha512 into _sha2. #101924
- PR: gh-99108: Import MD5 and SHA1 from HACL* #102089
- PR: gh-99108: Followup fix for Modules/Setup #102183
- PR: gh-99108: Add missing md5/sha1 defines to Modules/Setup #102308
- PR: gh-99108: Initial import of HACL-SHA3 into Python #103597
- PR: gh-99108: fix typo in Modules/Setup #104293
- PR: gh-99108: Refresh HACL* from upstream #104401
- PR: gh-99108: Release the GIL around hashlib built-in computation #104675
- PR: [3.12] gh-99108: Release the GIL around hashlib built-in computation (GH-104675) #104776
- PR: gh-99108: Refresh HACL* #104808
- PR: [3.12] gh-99108: Refresh HACL* (GH-104808) #104893
- PR: gh-99108: Mention HACL* in the hashlib docs. #105634
- PR: [3.12] gh-99108: Mention HACL* in the hashlib docs. (GH-105634) #105635

Background: HACL*

- Integrated in **Linux, Firefox, Tezos**, and many more
- **140,000+ lines** of verified F* code compiling to 80,000+ lines of C
- **30+ algorithms** and counting
- Proof engineer **productivity** is paramount

From Verified Crypto to “Real-World” Software

- HACL* is distributed as **C code**: non-negotiable, for perf.
- We issue a **PR** to “land” new HACL* algorithms **into** a project
- Project owner **reads the generated code**, audits, comments
- Usually, a **back-and-forth** to reach mutual satisfaction

```
uint32_t hLen = hash_len(a);
KRML_CHECK_SIZE(sizeof(uint8_t), hLen);
uint8_t m1Hash[hLen];
memset(m1Hash, 0U, hLen * sizeof(uint8_t));
uint32_t m1Len = (uint32_t)8U + hLen + saltLen;
KRML_CHECK_SIZE(sizeof(uint8_t), m1Len);
uint8_t m1[m1Len];
memset(m1, 0U, m1Len * sizeof(uint8_t));
memcpy(m1 + (uint32_t)8U, msg, msgLen);
```

jschanck

Should the patch add a check that `msgLen == hLen`?

The Python Challenge

hashlib:

- built-in library of hash functions
- a hodge-podge of implementations, all exposing the same API
- 5 variations of a similar state machine with an internal buffer
- could we factor out this redundancy?

Can we verify this code generically, *and* compile it to specialized C code?

When one thinks of genericity:

- OCaml: functors
- Haskell: typeclasses
- C++: templates
- …

Encoding Functors: Associative List Example

OCaml:

```
module type Map = sig
  type k
  val find: k -> (k * 'a) list -> 'a option
end

module type EqType = sig
  type t
  val eq: t -> t -> bool end

module MkMap (E : EqType) :
  Map with type k = E.t = struct
  type k = E.t
  let find x ls =
    let b = ref true in
    let lsp = ref ls in
    while !b do
      match !lsp with
      | [] -> b := false
      | (x', _) :: tl ->
        if E.eq x x' then b := false
        else lsp := tl done;
    match !lsp with
    | [] -> None
    | (_, y) :: _ -> Some y
end
```

Type constraint

We want a loop in the generated code

Doesn't compile to C (same with typeclasses)

F*:

Replace with a linked list

```
type map (a : Type) = {
  k: Type;
  find: k -> list (k * a) -> ST (option a) ... }

type eq_type = {
  t: Type;
  eq: t -> t -> bool; }

let mk_map (e : eq_type) (a : Type) :
  m:map a{m.k == e.t} = {
  k = e.t;
  find = (fun x ls ->
    let b = alloc true in
    let lsp = alloc ls in
    while (fun () -> !* b)
      (fun () ->
       let ls = !* lsp in
       match ls with
       | [] -> upd b false
       | (x', _) :: tl ->
         if e.eq x x' then upd b false
         else upd lsp tl);
    match !* lsp with
    | [] -> None
    | (_, y) :: _ -> Some y) }
```

Dictionary has runtime cost

Refinement

Proofs and annotations omitted

Extraction to C?

⇒ Specialization and partial evaluation?

Zero-Cost Functors: First Attempt (i)

Generic code (F*):

```
type map (a : Type) = {
  k: Type;
  find: k -> list (k * a) -> ST (option a) ... }

type eq_type = {
  t: Type;
  eq: t -> t -> bool; }

let mk_map (e : eq_type) (a : Type) :
  m:map a{m.k == e.t} = {
  k = e.t;
  find = (fun x ls ->
    let b = alloc true in
    let lsp = alloc ls in
    while (fun () -> !* b)
      (fun () ->
       let ls = !* lsp in
       match ls with
       | [] -> upd b false
       | (x', _) :: tl ->
         if e.eq x x' then upd b false
         else upd lsp tl);
    match !* lsp with
    | [] -> None | (_, y) :: _ -> Some y) }
```

Specialization:

```
let str_eqty : eq_type = { t = string; eq = String.eq; }
let ifind = (mk_map str_eqty int).find
```

After partial evaluation: Types are specialized

```
let ifind (x: string) (ls: list (string * int)) option int =
  let b = alloc true in let lsp = alloc ls in
  while (fun () -> !* b)
    (fun () ->
     let ls = !* lsp in
     match ls with
     | [] -> upd b false
     | (x', _) :: tl ->
       if String.eq x x'
       then upd b false
       else upd lsp tl);
  match !* lsp with
  | [] -> None | (_, y) :: _ -> Some y
```

e.eq is inlined

What happens if the code has several layers?

Zero-Cost Functors: First Attempt (ii)

Peer device for a secure channel protocol:

```
(* "Module signature" *)
type dv = {
  pid : Type;
  send : pid -> list (pid * ckey) -> bytes -> option bytes;
  recv : pid -> list (pid * ckey) -> bytes -> option bytes; }
```

```
(* "Module implementation" *)
type cipher = {
  enc : ckey -> bytes -> bytes;
  dec : ckey -> bytes -> option bytes; }

let mk_dv (m : map ckey) (c : cipher) : d:dv{d.pid == m.k} = {

  pid = m.k;

  send = (fun id dv plain ->
    match m.find id dv with
    | None -> None
    | Some sk -> Some (c.enc sk plain));

  recv = (fun id dv secret ->
    match m.find id dv with
    | None -> None
    | Some sk -> c.dec sk secret)
}
```

find gets inlined and duplicated

Zero-Cost Functors: Encoding

Parameterize with eq

```
(* Inline mk_find *)
let mk_find (k v : Type) (eq: k -> k -> bool) (x: k) (ls: list (k * v)) : option v =
  let b = alloc true in let lsp = alloc ls in
  while (fun () -> !* b)
    (fun () -> let ls = !* lsp in
    match ls with | [] -> upd b false
    | (x', _) :: tl -> if eq x x' then upd b false else upd lsp tl);
    match !* lsp with | [] -> None | (_, y) :: _ -> Some y)

(* Don't inline ifind *)
let ifind = mk_find i String.eq
```

Cumbersome to write and maintain

```
(* Inline mk_send *)
let mk_send (pid : Type) (find : pid -> list (pid * ckey) -> option ckey) (enc : ckey -> bytes -> bytes)
  (id : pid) (dv : list (pid * ckey)) (plain : bytes) : option bytes =
  match find id dv with
  | None -> None
  | Some sk -> Some (enc sk plain)

(* Don't inline isend *)
let isend = mk_send string ifind aes_enc

... (* mk_recv and irec *)
```

Zero-Cost Functors: Call-graph Rewriting

What we want to write:

```
type mindex = { k : Type; v : Type }

[@ Specialize]
assume val eq (i : mindex): i.k -> i.k -> bool

[@ Eliminate]
let while_cond (b: pointer bool) (_:unit) = !*b

[@ Eliminate]
let while_body (i: mindex) (b: pointer bool)
  (lsp: list (i.k * i.v)) (x:i.k) (_:unit) =
  let ls = !* lsp in
  match ls with
  | [] -> upd b false
  | (x', _) :: tl ->
    if eq x x' then upd b false
    else upd lsp tl

[@ Specialize]
let find (i : mindex) (x : i.k)
  (ls : list (i.k * i.v)) : option i.v =
  let b = alloc true in
  let lsp = alloc ls in
  while (while_cond b) (while_body i b lsp x);
  match !* lsp with
  | [] -> None | (_, y) :: _ -> Some y
```

What we want to get:

```
type mindex = { k : Type; v : Type }

let mk_find (i: mindex) (eq: i.k-> i.k -> bool)
  (x: i.k) (ls: list (i.k * i.v)): option i.v =
  let b = alloc true in let lsp = alloc ls in
  while (fun () -> !* b)
    (fun () -> let ls = !* lsp in
     match ls with
     | [] -> upd b false
     | (x', _) :: tl ->
       if eq x x' then upd b false
       else upd lsp tl);
  match !* lsp with
  | [] -> None
  | (_, y) :: _ -> Some y
```

The code is re-checked

Similar (more complex) device used in the Noise* protocol compiler

```
%splice [ mk_find ] (specialize (`mindex) [`find ])
```

Call-graph rewriting by means of meta-programming

Application: algorithms in HACL*

- Type parameter = choice of vectorization level (None, 128-bit, 256-bit)
- Code = crypto algorithm, e.g. Chacha20, Poly1305, etc.
- Deep static call graphs, mixture of [@ Specialize] and [@ Eliminate]

```
module Hacl.Meta.Chacha20Poly1305

#set-options "--z3rlimit 350 --max_fuel 0 --max_ifuel 1"

%splice[
  mk_chacha20poly1305_poly1305_do;
  mk_chacha20poly1305_aead_encrypt;
  mk_chacha20poly1305_aead_decrypt
] (Meta.Interface.specialize (`Hacl.Impl.Poly1305.Fields.field_spec) [
    `Hacl.Impl.Chacha20Poly1305.aead_encrypt;
    `Hacl.Impl.Chacha20Poly1305.aead_decrypt
])

```

```
let aead_decrypt : aead_decrypt_st M256 =
  mk_chacha20poly1305_aead_decrypt #M256 True Hacl.Chacha20.Vec256.chacha20_encrypt_256 poly1305_do_256
```

Application: algorithms in HACL*

Algorithm	Number of specializations	Nature of specialization
Chacha20	3	vectorization level
Poly1305	3	vectorization level
Chacha20Poly1305	3	vectorization level
HPKE	15 (> 80 possible options)	ciphersuite & implementation
Curve25519	3 (recursive)	field arithmetic

Curve25519 has two recursive layers of specialization:

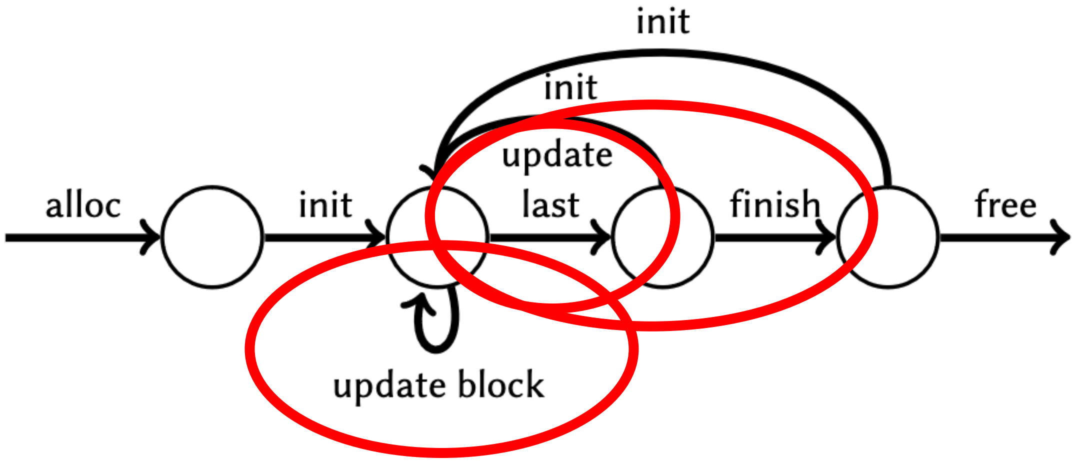
- Field64 can be specialized with Vale (ASM) or HACL (C)
- Curve25519 itself can be specialized with Field64 or Field51

All those implementations: > 20k lines of C code

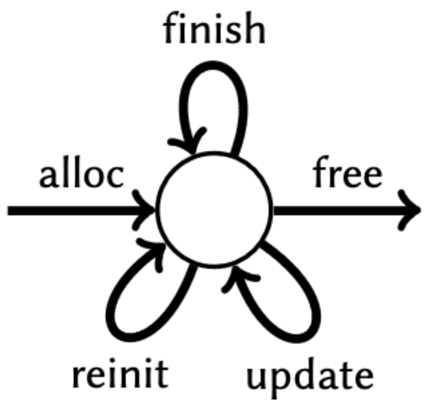
Application: Streaming APIs

Consider a ***block*** API, such as a ***hash function:***

- tricky state machine
- must feed data in entire blocks (unrealistic)
- computing the hash invalidates the state
- precise sequence of operations



Application: Streaming APIs

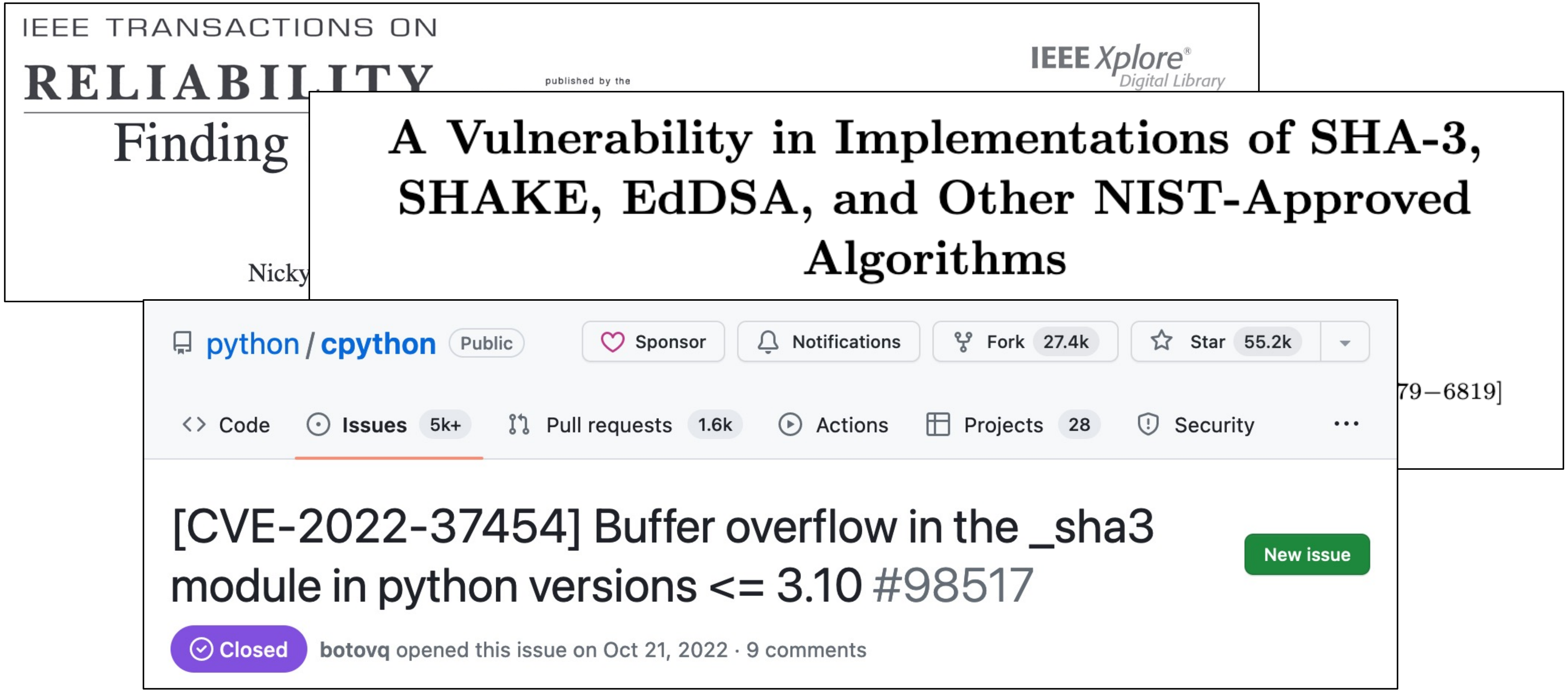


Instead, people use ***Streaming APIs***:

- Long-lived state carries internal buffer
- Incremental "update" operation ***accumulates*** arbitrary-sized data
- Intermediary digests do not invalidate the state
- Internal details such as update_last are hidden
- **Tricky to implement correctly**

```
state *s = hash_new(SHA2_256);
hash_update(s, "hello", 5);
hash_update(s, " ", 1);
char hash1[32];
hash_digest(s, hash1);
char hash2[32];
hash_update(s, "world", 5);
hash_digest(s, hash2);
hash_delete(s);
```

Application: Streaming APIs



Application: Streaming APIs

- Flagship application of our techniques!
- Use the rewriting tactic and earlier code patterns to write ***one streaming API*** that is ***generic over the choice of block algorithm***
- ***Enormous*** code savings:
  - 15 applications of the generic code
  - really common API! (though many tweaks: key/ no key, runtime key, etc.)
  - proof-to-code ratio of 0.51: every line of F* yields two lines of C code (total: 8k)
  - in relative terms: massive improvement compared to earlier versions of HACL

Application: Streaming APIs

Extract from one of the "index" types:

```
inline_for_extraction noextract noeq
type block (index: Type0) =
| Block:
  km: key_management ->

  // Low-level types
  state: stateful index ->
  key: stateful index ->

  // Just a value type; useful for algorithms that have a variable output length.
  output_length_t: Type0 ->

  // Introducing a notion of blocks and final result.
  max_input_len: (index -> x:U64.t { U64.v x > 0 }) ->
  output_length: (index -> output_length_t -> GTot Lib.IntTypes.(x:size_nat { x > 0 })) ->
  block_len: (index -> x:U32.t { U32.v x > 0 }) ->
  // The size of data to process at a time. Must be a multiple of block_len.
  // Controls the size of the internal buffer.
  blocks_state_len: (i:index ->
    x:U32.t { U32.v x > 0 /\ U32.v x >= U32.v (block_len i) /\ U32.v x % U32.v (block_len i) = 0 }) ->
  init_input_len: (i:index -> x:U32.t { U32.v x <= U32.v (block_len i) /\ U32.v x <= U64.v (max_input_len i) }) ->

  /// An init/update/update_last/finish specification. The long refinements were
  /// previously defined as blocks / small / output.
  init_input_s: (i:index -> key.t i -> s:S.seq uint8 { S.length s = U32.v (init_input_len i) }) ->
  init_s: (i:index -> key.t i -> state.t i) ->
  update_multi_s: (i:index ->
    state.t i ->
    prevlen:nat { prevlen % U32.v (block_len i) = 0 } ->
    s:S.seq uint8 { prevlen + S.length s <= U64.v (max_input_len i) /\ S.length s % U32.v (block_len i) = 0 } ->
    state.t i) ->
  update_last_s: (i:index ->
    state.t i ->
    prevlen:nat { prevlen % U32.v (block_len i) = 0 } ->
    s:S.seq uint8 { S.length s + prevlen <= U64.v (max_input_len i) /\ S.length s <= U32.v (block_len i) } ->
    state.t i) ->
  finish_s: (i:index -> key.t i -> state.t i -> l:output_length_t ->
    s:S.seq uint8 { S.length s = output_length i l }) ->
```

Types

Constants

Spec (spec defs + theorems)

Application: Streaming APIs

- Excellent engagement with the Python team
- Replaced all of their built-in hash implementations with our verified code
- Released in Python 3.12 (blake2 is coming)
- Good confirmation that our work has practical impact
- Forced us to polish, attain a high level of quality, and do serious packaging work

Modularity, Code Specialization, and Zero-Cost Abstractions for Program Verification

Son Ho (INRIA), Aymeric Fromherz (INRIA), Jonathan Protzenko (Microsoft Research)

- An arsenal of ***PL techniques*** to reconcile high-level, generic programming with low-level code specialization and verification ("best of both worlds")
- Added an ***extra compiler stage*** that automatically rewrites the user's code in userland
- Wide variety of ***applications in HACL***, significant boost on productivity, maintenance
- One ***flagship application***: streaming functor, an "API transformer" that goes from unsafe API to safe API, ***integrated into Python***