Formal Verification of SHA-3: Proof Techniques for Symmetric Cryptography

Tristan Schwieren

Technical University of Munich, Cryspen Sarl

Advisor: Karthikeyan Bhargavan

Supervisor: Julian Pritzi

Cryptography is everywhere

Cryptography is the foundation of secure systems…

and

…single bug in cryptography can break a systems security completely

Bugs in Cryptography

- Most attacks on cryptographic systems exploit bugs in implementation, not flaws in their design
    - One vulnerability per 700 - 1700 lines changed
- **We aim for guarantees not just confidence**
- We argue: Using Formal Methods has lower cost than patches post deployment

Collaboration with ~~CRYSPEN~~

- Cryspen is a Startup that builds High Assurance Software
- HAX: Rust verification Framework
  - State of the art in high assurance software
  - **Make verification easier for non formal methods experts**
  - Translate (a subset of) Rust to formal languages (F*, Rocq, Lean)
- libcrux: Free and open source, fast, portable and formally verified cryptographic library written in rust 🦀

HAX is the tool needed to build libcrux

Research Gap / Problem statement

Can HAX formally verify algorithms such as SHA-3?

- Heavier use of bit-level operations (shifts, rotations, XORs)
- Round based permutations
- Loop over arbitrary input and handling buffers

SHA-3 reference implementation was vulnerable! (CVE-2022-37454)

Evaluate and improve the HAX tool chain for modern symmetric cryptography

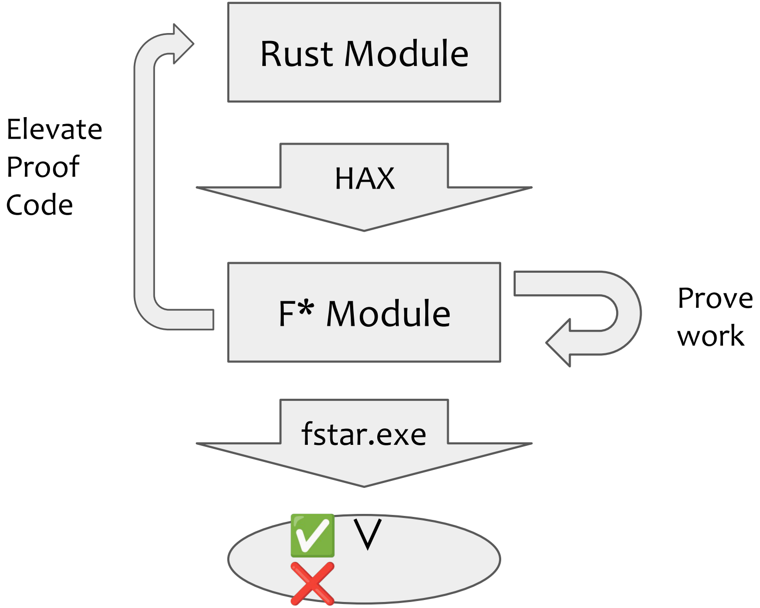
Outline

- ~~Overview~~
- Background
- Design & Implementation
- Evaluation
- Conclusion

Background: What and how to prove things?

Prove properties of you program!

- Memory safety
- Panic Freedom
- Termination
- Correctness
- …



Overview: Example

```rust
fn set(
    arr: &mut [u64; 25],
    x: usize,
    y: usize,
    value: u64,
) {



    arr[5 * y + x] = value;
}
```

Background: Example

```rust
fn set(
    arr: &mut [u64; 25],
    x: usize,
    y: usize,
    value: u64,
) {

    debug_assert!(x < 5 && y < 5);

    arr[5 * y + x] = value;
}
```

Background: Example

```rust
#[hax::requires(x < 5 && y < 5)]
fn set(
    arr: &mut [u64; 25],
    x: usize,
    y: usize,
    value: u64,
) {

    debug_assert!(x < 5 && y < 5);

    arr[5 * y + x] = value;
}
```

Background: Example

```rust
#[hax::requires(x < 5 && y < 5)]
fn set(
    arr: &mut [u64; 25],
    x: usize,
    y: usize,
    value: u64,
) {
    #[cfg(not(hax))]
    debug_assert!(x < 5 && y < 5);

    arr[5 * y + x] = value;
}
```

Background: Example

```
let set
      (arr: t_Array u64 (mk_usize 25))
      (x y: usize)
      (value: u64)
    : Prims.Pure (t_Array u64 (mk_usize 25))
      (requires x <. mk_usize 5 && y <. mk_usize 5)
      (fun _ -> Prims.l_True) =
 let arr = update_at_usize arr ((mk_usize 5 *! y) +! x) value in
 arr
```

```
i: usize { v i < Seq.length arr }
```

Outline

- ~~Overview~~
- ~~Background~~
- Design & Implementation
- Evaluation
- Conclusion

Design: libcrux-sha3

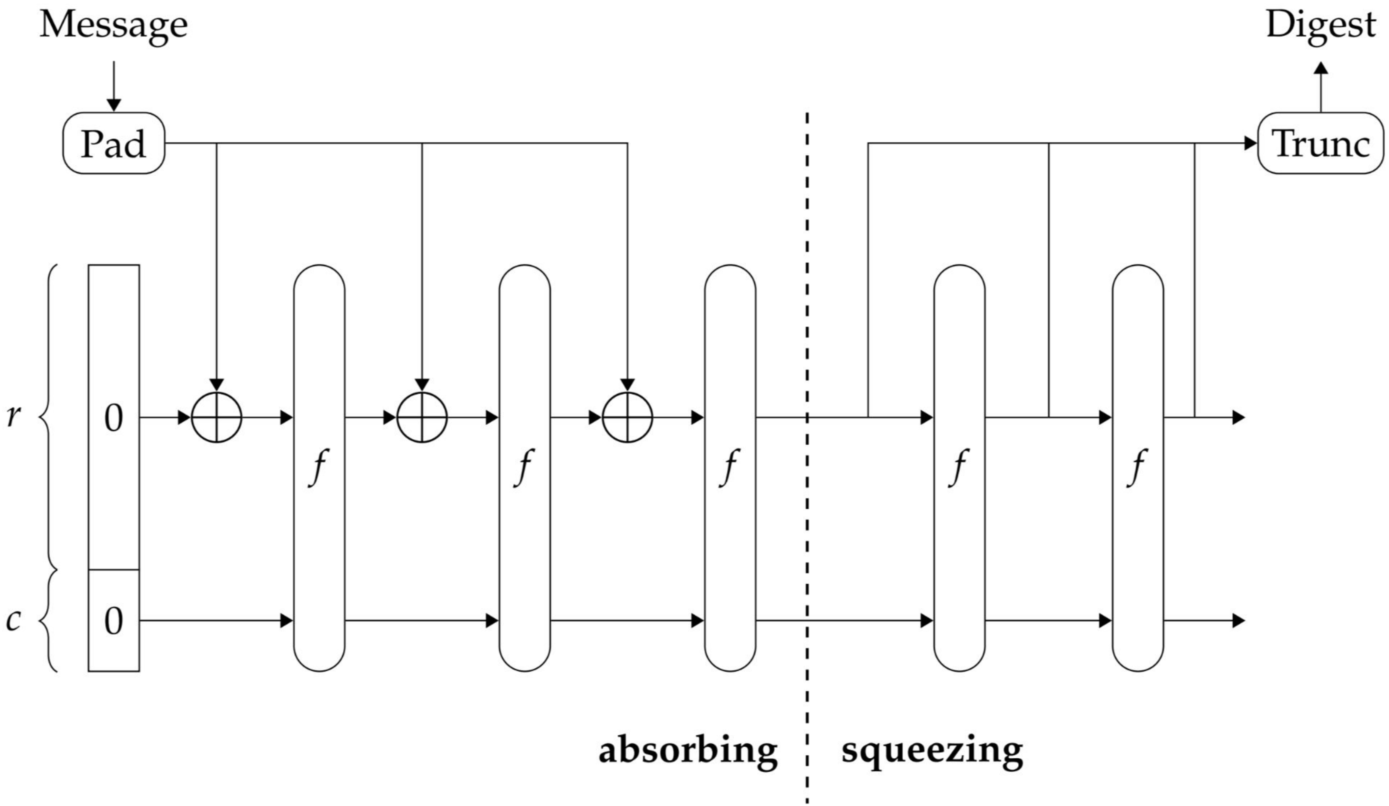
- Implements all versions of the SHA-3 Standard
  - SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, SHAKE256
- Different backends
  - Portable → Hash a single input
  - NEON → Hash two inputs
  - AVX2 → Hash four inputs
- **Buffering API to incrementally absorb inputs**

Design: Buffering XOF API

```
trait Xof {
    /// Create new absorb state
    fn new() -> Self;

    /// Absorb input
    fn absorb(&mut self, input: &[u8]);

    /// Absorb final input (may be empty)
    fn absorb_final(&mut self, input: &[u8]);

    /// Squeeze output bytes
    fn squeeze(&mut self, out: &mut [u8]);
}
```

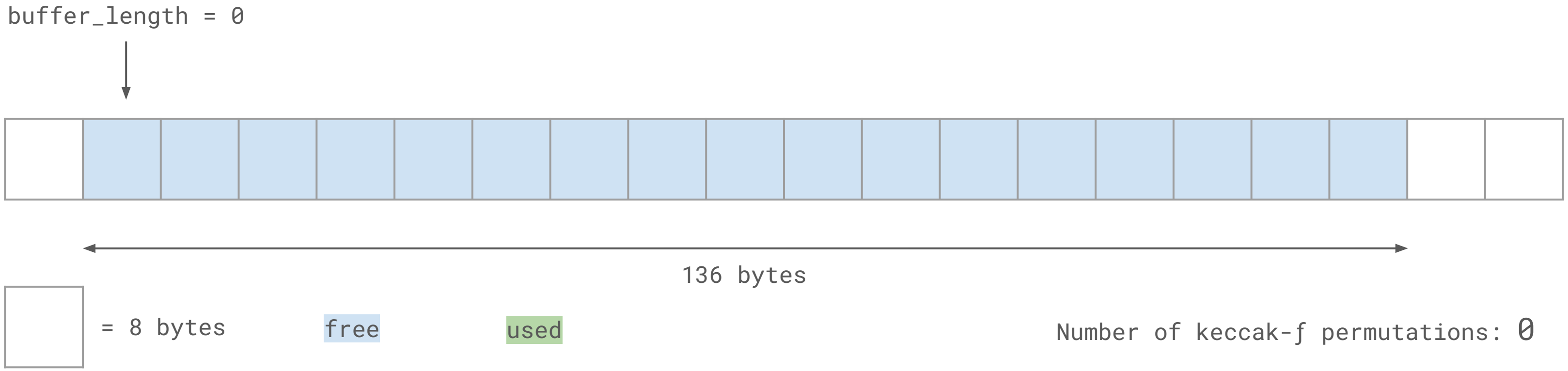
Be able to iteratively absorb and squeeze any number of bytes

Design: Sponge Construction



Implementation: XOF Buffer API

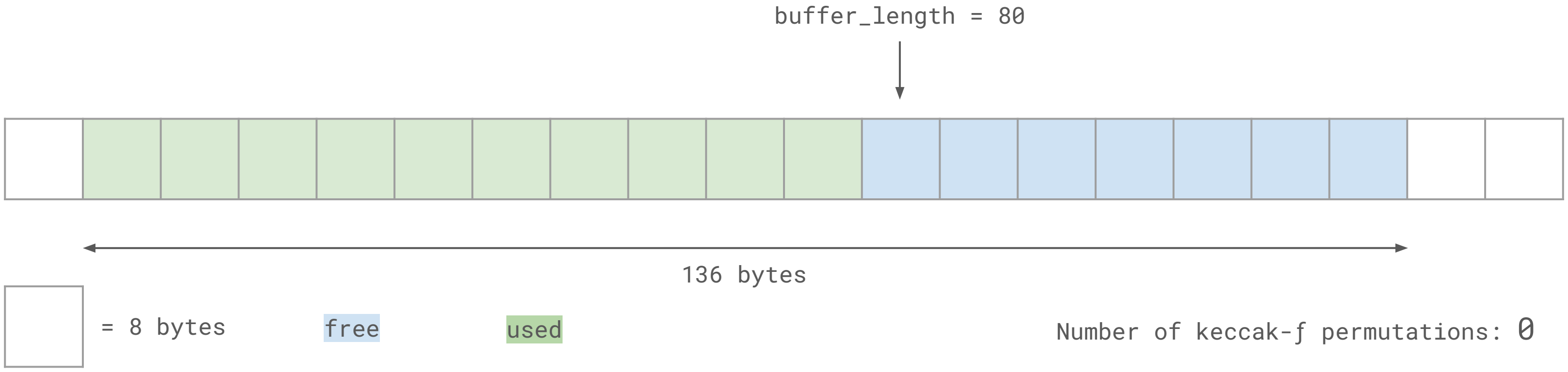
```rust
let mut state = Shake256Xof::new();
let mut out = [0u8; 32];
```



Number of keccak-f permutations: 0

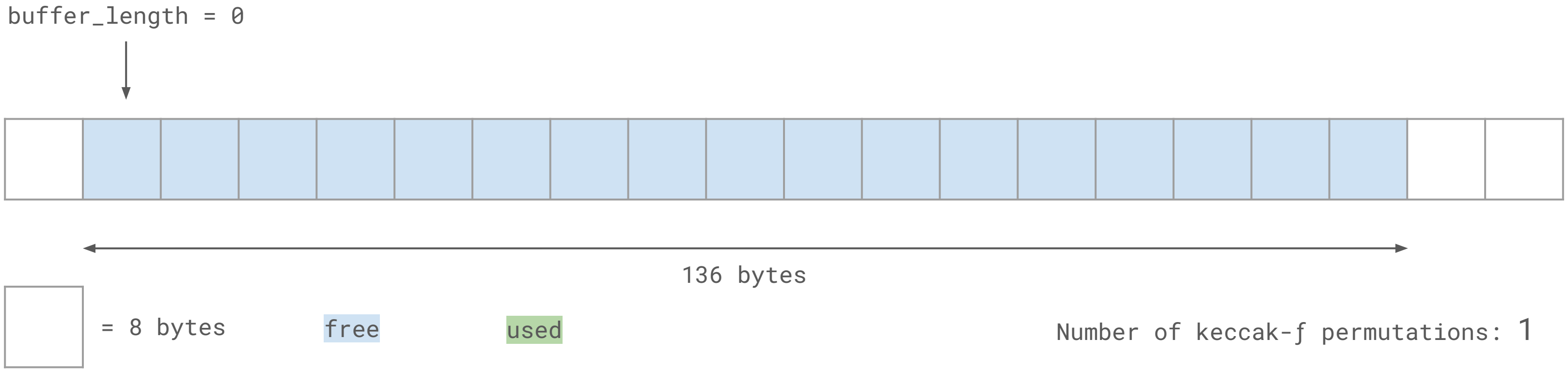
Implementation: XOF Buffer API

```
let mut state = Shake256Xof::new();
let mut out = [0u8; 32];

state.absorb(&input[..80]);
```



Implementation: XOF Buffer API

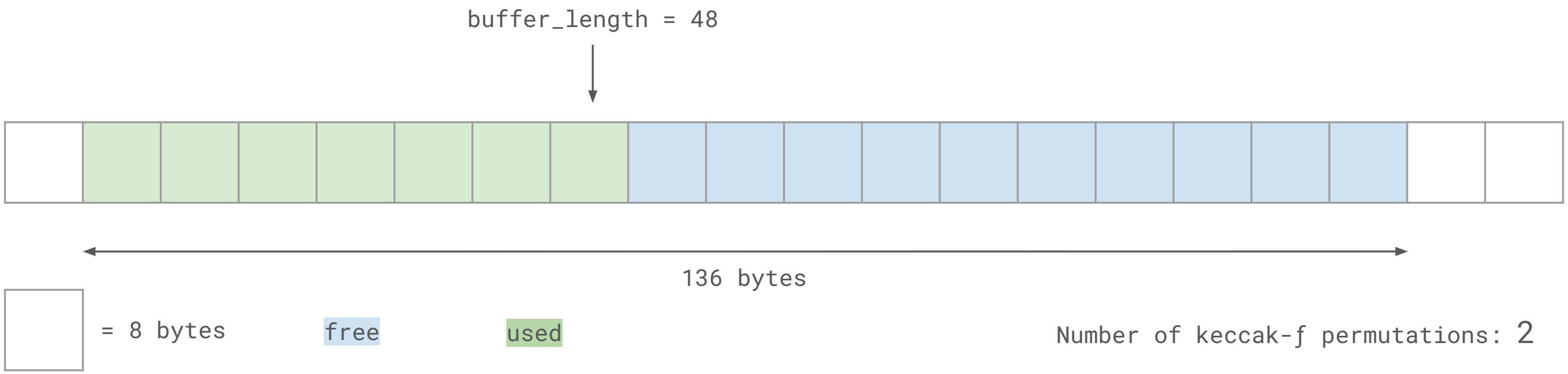
```rust
let mut state = Shake256Xof::new();
let mut out = [0u8; 32];

state.absorb(&input[..80]);
state.absorb(&input[80..136]);
```



Number of keccak-f permutations: 1

Implementation: XOF Buffer API

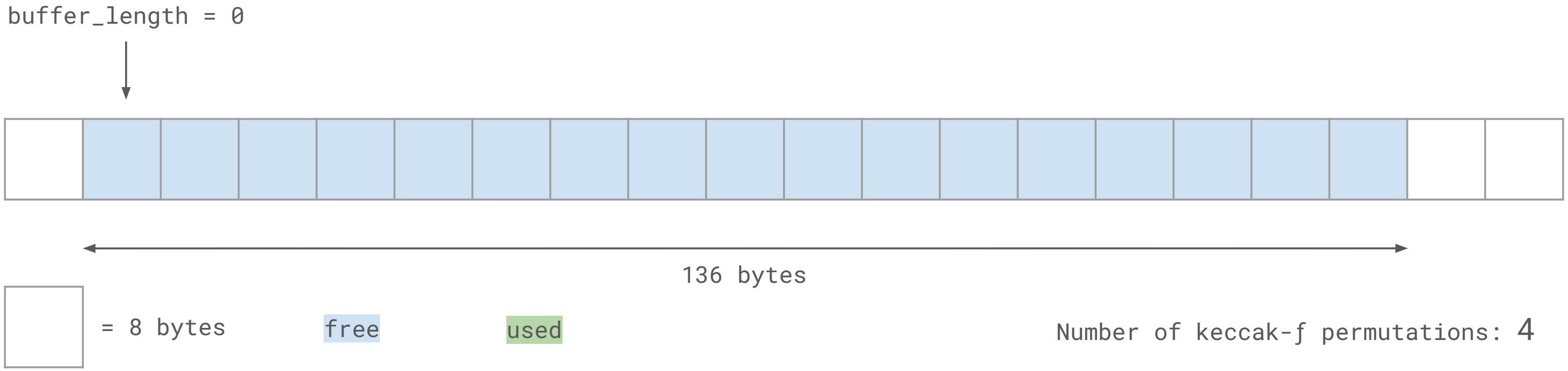
```
let mut state = Shake256Xof::new();
let mut out = [0u8; 32];

state.absorb(&input[..80]);
state.absorb(&input[80..136]);
state.absorb(&input[136..320]);
```



Number of keccak-f permutations: 2

Implementation: XOF Buffer API

```
let mut state = Shake256Xof::new();
let mut out = [0u8; 32];

state.absorb(&input[..80]);
state.absorb(&input[80..136]);
state.absorb(&input[136..320]);
state.absorb_final(&input[320..408]);
```



Number of keccak-f permutations: 4

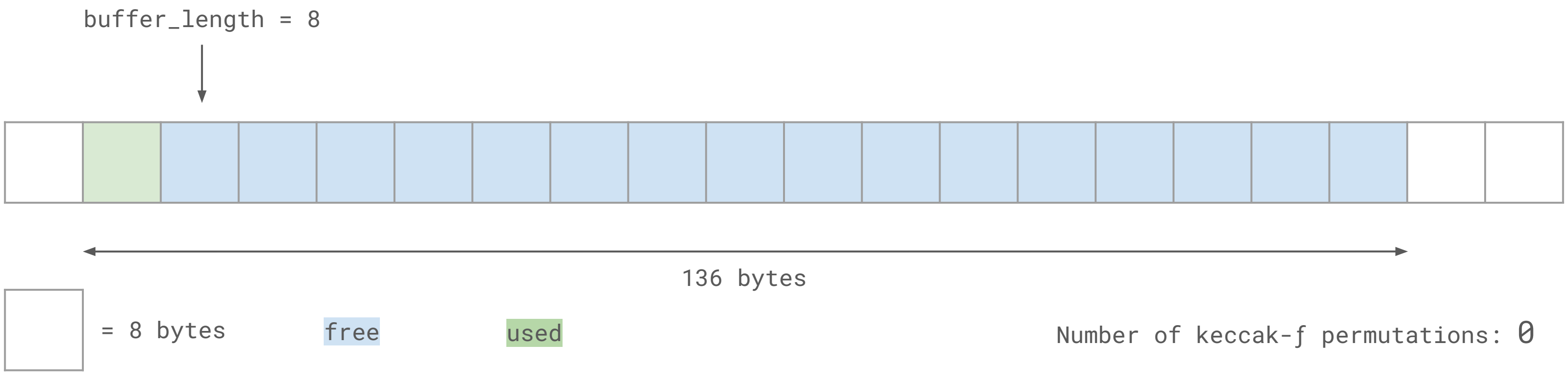
Implementation: Additional absorb features

```
pub fn fill_buffer(&mut self, inputs: &[u8]) -> usize
```

Fill the internal buffer and return the absorbed bytes

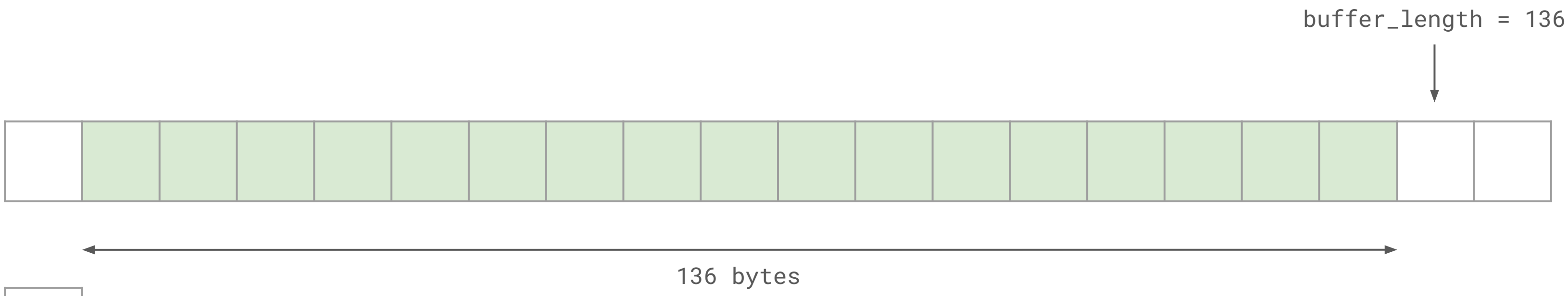
Implementation: XOF Buffer API Panic

```
let mut state = Shake256Xof::new();
let mut out = [0u8; 32];

state.absorb(&INPUT[..8]);
```



Implementation: XOF Buffer API Panic

```rust
let mut state = Shake256Xof::new();
let mut out = [0u8; 32];

state.absorb(&INPUT[..8]);
let rem = state.fill_buffer(&INPUT[8..136]);
assert!(128 == rem);
```

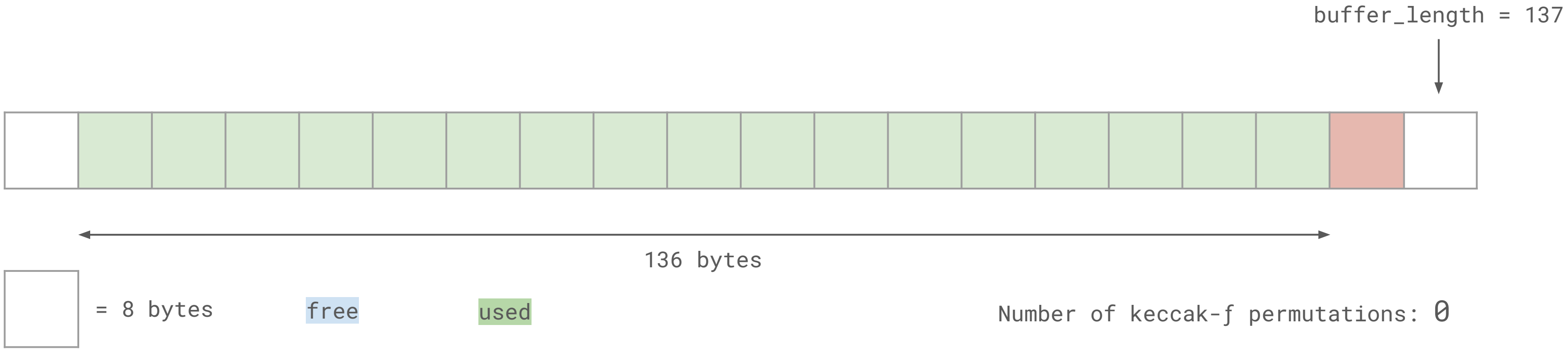


= 8 bytes   free   used

Number of keccak-f permutations: 0

Implementation: XOF Buffer API Panic

```rust
let mut state = Shake256Xof::new();
let mut out = [0u8; 32];

state.absorb(&INPUT[..8]);
let rem = state.fill_buffer(&INPUT[8..136]);
assert!(0 == rem);
state.absorb(&INPUT[136..137]);
```



Implementation: XOF Buffer API Panic

```
let consumed = self.fill_buffer(input);

if consumed > 0 {
if self.buffer_length == RATE {
  self.state.absorb_block::<RATE>(&self.buffer, 0);
  self.state.keccakf1600();
  self.buffer_length = 0;
}
```

Beginning of the absorb function

Implementation: State invariants

```
impl Xof for Shake256Xof {
  #[hax_lib::ensures(|result| xof_state_inv(&result))]
  fn new() -> Self

  #[hax_lib::requires(xof_state_inv(&self))]
  #[hax_lib::ensures(|_| xof_state_inv(&future(self)))]
  fn absorb(&mut self, input: &[u8])

  #[hax_lib::requires(xof_state_inv(&self))]
  #[hax_lib::ensures(|_| xof_state_inv(&future(self)))]
  fn absorb_final(&mut self, input: &[u8])

  #[hax_lib::requires(xof_state_inv(&self))]
  #[hax_lib::ensures(|_| xof_state_inv(&self))]
  fn squeeze(&mut self, out: &mut [u8])
}
```

```
fn xof_state_inv (
   xof: &Shake256Xof
) -> bool {
   xof.buffer_length <= 136
}
```

Outline

- ~~Overview~~
- ~~Background~~
- ~~Design & Implementation~~
- Evaluation
- Conclusion

Evaluation

- **Formal Verification of industry grade SHA-3 Implementation**
- **Formal Verification with HAX still is a very manual task**
- HAX Limitations:
  - Large Permutations cause exponential blowup in verification time
  - Types with mutable references don’t work ⟶ No `&[&mut [u8]; N]`
  - Core library misses bit manipulations ⟶ No correctness proof

```
u64::rotate_left(x, LEFT)
u64::rotate_right(x, RIGHT)
x << LEFT
x >> RIGHT
```

```
trait Squeeze<N: usize> {
    fn squeeze(&self, out: &[&mut [u8]; N]);
}
```

Outline

- ~~Overview~~
- ~~Background~~
- ~~Design & Implementation~~
- ~~Evaluation~~
- Conclusion

Conclusion

HAX and Formal Verification are useful for finding edge case bugs in high assurance software!

Conclusion: Impact

Merged Libcrux Pull Requests:
- Add tests for hash functions #981
- Fixes the flake to work with rust-rover IDE #994
- Sha3 lax check #1092
- Makefile improvement #1101
- Remove unsafe expression on safe AVX2 #1109
- SHA3 SIMD tests #1120
- bump HAX version #1127
- Fix warning. Remove unused lemma #1145

Open Libcrux Pull Requests:
- Add full type check for SHA-3 portable #1157
- Panic freedom for SIMD backends #1238

Merged HAX Pull Requests:
- Fix type of u64 rotate left #1548
- fix Core.Clone #1552
- rename explicit_panic #1605
- Change impl_u64__rotate_right second parameter type to u32 #1778

libcrux-sha3 is used in Signal's new Post-Quantum Ratchet!

Outline

- ~~Overview~~
- ~~Background~~
- ~~Design & Implementation~~
- ~~Evaluation~~
- ~~Conclusion~~

Questions