Survey of Verified Cryptography

CSCI-762 / Tom Arnold

Verified Cryptography

- Using tools to apply cryptography and prove that the implementation is correct
- Tricky implementation details
- Assumptions: algorithm is correct, cryptographic primitives can be verified and then built on

Memory Safety

- High performance code usually written in memory unsafe languages (C/C++).
- Vulnerabilities can allow attacker can access arbitrary memory.
    - OpenSSL Heartbleed - Heartbeat request returning uninitialized memory to client, allowing client (slow) read access to memory to hunt for private keys and other secrets
- Managed languages like Java solve this but have worse performance, complex implementation, unusable for embedded/legacy code bases

Functional Correctness

- Specification usually given as IETF (Internet Task Force) RFC document
  - Turn spec into code, does it still match spec?
  - Optimize code, does it still match spec?
- Exhaustive test suite not trivial
  - Property based testing
  - Proof assistants

Side-channel Resistance

- Attacker can determine information by observing runtime, CPU usage, power usage, etc

- Can be caused by optimizations in code, e.g. shortcut multiplication when operand is zero

- Can be caused by arch level (CPU/memory) optimizations like branch prediction and cache

- Secret independence - Don't allow optimizations/ shortcuts based on secret values

FStar (F*) Programming Language

- ML-based programming language from INRIA and Microsoft Research

- Features: refinement types, dependent types, proof assistant

- Pre-post conditions on functions that the compiler can use to prove the code is correct

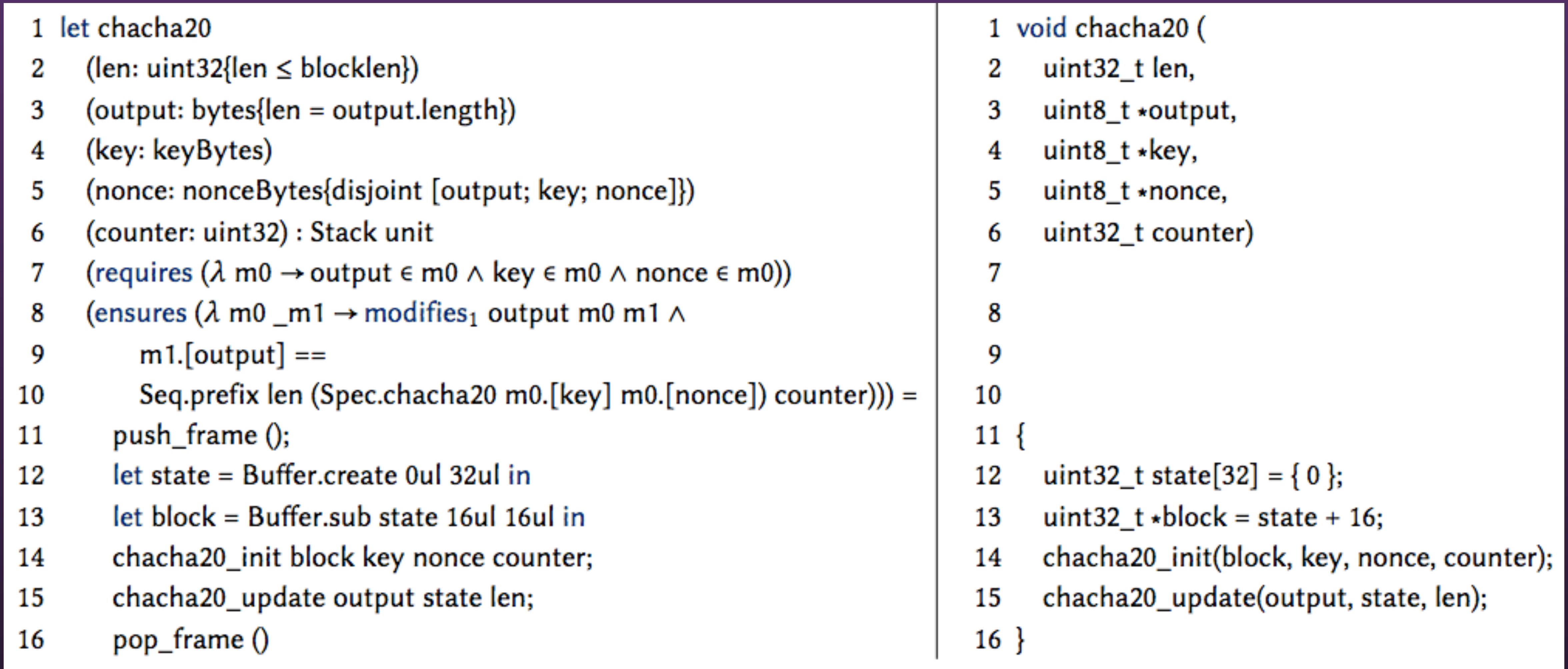
- Compiles to OCaml/F#

- Low* dialect compiles to readable C code

F*/Low* Approach

- Executable specifications (proofs) written in high-level F*, operations written in Low* dialect
  - No recursive data structures, no dynamic allocation, bounded heaps (region based memory management)
- Low* compiled to C for inclusion in other software or manual verification
- Use the powerful type system to enforce memory safety and secret independence in generated code
  - Abstract type for secret integers that only allows constant time operations

Example - ChaCha20



```
let chacha20
  (len: uint32{len ≤ blocklen})
  (output: bytes{len = output.length})
  (key: keyBytes)
  (nonce: nonceBytes{disjoint [output; key; nonce]})
  (counter: uint32) : Stack unit
  (requires (λ m0 → output ∈ m0 ∧ key ∈ m0 ∧ nonce ∈ m0))
  (ensures (λ m0 _m1 → modifies₁ output m0 m1 ∧
      m1.[output] ==
      Seq.prefix len (Spec.chacha20 m0.[key] m0.[nonce]) counter))) =
    push_frame ();
    let state = Buffer.create 0ul 32ul in
    let block = Buffer.sub state 16ul 16ul in
    chacha20_init block key nonce counter;
    chacha20_update output state len;
    pop_frame ()
```

```
void chacha20 (
  uint32_t len,
  uint8_t *output,
  uint8_t *key,
  uint8_t *nonce,
  uint32_t counter)




{
  uint32_t state[32] = { 0 };
  uint32_t *block = state + 16;
  chacha20_init(block, key, nonce, counter);
  chacha20_update(output, state, len);
}
```

Fig. 2. A snippet from ChaCha20 in Low* (left) and its C compilation (right)

https://arxiv.org/pdf/1703.00053.pdf

HACL*

- Library of verified cryptography primitives written in F*
  - Stream ciphers: ChaCha20, Salsa20
  - Hashing: SHA-2
  - Signature: Ed25519
  - Authentication: Poly1305, HMAC-SHA-2
  - Authenticated crypto: ChaCha20-Poly1305
- Performance between OpenSSL C and ASM
- Proof-to-code ratio 2:1

References

- HACL*: A Verified Modern Cryptographic Library: https://eprint.iacr.org/2017/536.pdf
- Verified Low Level Programming Embedded in F*: https://arxiv.org/pdf/1703.00053.pdf