High-assurance crypto in practice – Challenges and latest results

Peter Schwabe
September 11, 2023
Make crypto software boring again

Peter Schwabe
September 11, 2023
The setting for “boring” crypto software

- Primitives, no protocols
- “Secure-channel” primitives
The setting for "boring" crypto software

- Primitives, no protocols
- "Secure-channel" primitives
- Only software-visible side channels
The setting for “boring” crypto software

- Primitives, no protocols
- “Secure-channel” primitives
- Only software-visible side channels
- Big CPUs
Back in the days...

- Use X25519, Ed25519
- Use SHA2, ChaCha20, Poly1305
Back in the days...

- Use X25519, Ed25519 (or NISTP256-ECDH, ECDSA)
- Use SHA2, ChaCha20, Poly1305 (or AES, HMAC)
Back in the days...

- Use X25519, Ed25519 (or NISTP256-ECDH, ECDSA)
- Use SHA2, ChaCha20, Poly1305 (or AES, HMAC)
- Follow “constant-time” paradigm
  - No secret-dependent branches
  - No memory access at secret-dependent location
  - No variable-time arithmetic (e.g., DIV)
• Use X25519, Ed25519 (or NISTP256-ECDH, ECDSA)
• Use SHA2, ChaCha20, Poly1305 (or AES, HMAC)
• Follow “constant-time” paradigm
  • No secret-dependent branches
  • No memory access at secret-dependent location
  • No variable-time arithmetic (e.g., DIV)
• Fairly little code, doesn’t even need function calls!
• More assumptions, more schemes, more parameters, *more code*
• More complexity in implementations, protocols, and proofs
• More assumptions, more schemes, more parameters, more code
• More complexity in implementations, protocols, and proofs
• Initially many bugs that were not caught by functional testing
• Early personal intuition:
  • no big-integer arithmetic $\rightarrow$ no “rare” bugs
  • Confidence in functional correctness through test vectors . . . ?

Shattered by Hwang, Liu, Seiler, Shi, Tsai, Wang, and Yang (CHES 2022): Verified NTT Multiplications for NIST PQC KEM Lattice Finalists: Kyber, SABER, and NTRU.
• More assumptions, more schemes, more parameters, **more code**
• More complexity in implementations, protocols, and proofs
• Initially many bugs that were not caught by functional testing
• Early personal intuition:
  • no big-integer arithmetic → no “rare” bugs
  • Confidence in functional correctness through test vectors . . . ?
• Shattered by Hwang, Liu, Seiler, Shi, Tsai, Wang, and Yang (CHES 2022): **Verified NTT Multiplications for NISTPQC KEM Lattice Finalists: Kyber, SABER, and NTRU.**
Advanced microarchitectural side channels
Advanced microarchitectural side channels

https://www.metal-archives.com/bands/Downfall/3540377075
Who here has written some crypto software?
Who here has written some crypto software?

Who used C?
What’s wrong with C?

- No memory safety
- Finicky semantics
  - Undefined behavior
  - Implementation-specific behavior
  - Context-specific behavior
- No mandatory initialization
- No (optional) runtime checks
What’s wrong with C?

• No memory safety
• Finicky semantics
  • Undefined behavior
  • Implementation-specific behavior
  • Context-specific behavior
• No mandatory initialization
• No (optional) runtime checks

but... Rust!

• Memory safe
• More clear semantics (?)
• Mandatory variable initialization
• (Optional) runtime checks for, e.g., overflows
Lack of security features

“Security engineers have been fighting with C compilers for years.”

—Simon, Chisnall, Anderson, 2018

• No concept of secret vs. public data
• No preservation of “constant-time”
• No (or very limited) protection against microarchitectural attacks
• No erasure of sensitive data

1 What you get is what you C: Controlling side effects in mainstream C compilers. EuroS&P 2018
Let’s fix those tools!

“We argue that we must stop fighting the compiler, and instead make it our ally.”

—Simon, Chisnall, Anderson, 2018
Let's fix those tools!

Secure erasure in LLVM

- Simon, Chisnall, Anderson implement secure erasure in LLVM
- Code available at https://github.com/lmrs2/zerostack
- Not adopted in mainline LLVM
Let’s fix those tools!

Secret types in Rust + LLVM

- Initiative at HACS 2020: secret integer types in Rust, C++, and LLVM
- Rust draft RFC online at https://github.com/rust-lang/rfcs/pull/2859
- Implementation in LLVM is massive effort, no real progress, yet
Let's fix those tools!

Spectre protections in LLVM

- Carruth, 2019: Spectre v1 countermeasure in LLVM\(^2\) (see later in the talk)
- "does not defend against secret data already loaded from memory and residing in registers"

---

\(^2\)[https://llvm.org/docs/SpeculativeLoadHardening.html]

\(^3\)Ultimate SLH: Taking Speculative Load Hardening to the Next Level. USENIX Security, 2023
Let’s fix those tools!

Spectre protections in LLVM

- Carruth, 2019: Spectre v1 countermeasure in LLVM\(^2\) (see later in the talk)
- “does not defend against secret data already loaded from memory and residing in registers”
- Zhang, Barthe, Chuengsatiansup, Schwabe, Yarom, 2023: More principled approach\(^3\)
- Report and proposed patches to LLVM in March 2022
- September 2022: **Status: WontFix (was: New)**

---

\(^2\)[https://llvm.org/docs/SpeculativeLoadHardening.html]

\(^3\) *Ultimate SLH: Taking Speculative Load Hardening to the Next Level*. USENIX Security, 2023
High-assurance crypto

FORMOSA CRYPTO

• Effort to formally verify crypto
• Goal: verified PQC ready for deployment
• Three main projects:
  • EasyCrypt proof assistant
  • Jasmin programming language
  • Libjade (PQ-)crypto library
• Core community of \( \approx 30-40 \) people
• Discussion forum with \( >180 \) people
Formosan black bear

The Formosan black bear (臺灣黑熊, Ursus thibetanus formosanus), also known as the Taiwanese black bear or white-throated bear, is a subspecies of the Asiatic black bear. It was first described by Robert Swinhoe in 1864. Formosan black bears are endemic to Taiwan. They are also the largest land animals and the only native bears (Ursidae) in Taiwan. They are seen to represent the Taiwanese nation.

Because of severe exploitation and habitat degradation in recent decades, populations of wild Formosan black bears have been declining. This species was listed as "endangered" under Taiwan's Wildlife Conservation Act (Traditional Chinese: 野生動物保育法) in 1989. Their geographic distribution is restricted to remote, rugged areas at elevations of 1,000–3,500 metres (3,300–11,500 ft). The estimated number of individuals is 200 to 600.[3]

Physical characteristics  [edit]

The Formosan black bear is sturdily built and has a round head, short neck, small eyes, and long snout. Its head measures 26–35 cm (10–14 in) in length and 40–60 cm (16–24 in) in circumference. Its ears are 8–12 cm (3.1–4.7 in) long. Its snout resembles a dog's, hence its nickname is "dog bear". Its tail is inconspicuous and short—usually less than 10 cm (3.9 in) long. Its body is well covered with rough, glossy, black hair, which can grow over 10 cm long around the neck. The tip of its chin is white. On the chest, there is a

https://en.wikipedia.org/wiki/Formosan_black_bear
High-assurance crypto

- Effort to formally verify crypto
- Goal: **verified PQC ready for deployment**
- Three main projects:
  - EasyCrypt proof assistant
  - Jasmin programming language
  - Libjade (PQ-)crypto library
- Core community of \( \approx 30–40 \) people
- Discussion forum with >180 people
Joint work with...

The toolchain and workflow
The toolchain and workflow

Interactive security proofs

EasyCrypt specification

EasyCrypt model of Jasmin program

Jasmin program

Automated checks

Certified compilation

Assembly
The toolchain and workflow

- **EasyCrypt specification**
- **EasyCrypt model of Jasmin program**
- **Jasmin program**
- **Jasmin Compiler**
- **Assembly**

Interactive security proofs

Interactive proof

Extracts to

Automated checks

Certified compilation
The toolchain and workflow

Hülsing, Meijers, and Strub. *Formal Verification of Saber’s Public-Key Encryption Scheme in EasyCrypt.* CRYPTO 2022

Hülsing, Meijers, and Strub. *Formal Verification of Saber’s Public-Key Encryption Scheme in EasyCrypt*. CRYPTO 2022

Barbosa, Barthe, Doczkal, Don, Fehr, Grégoire, Huang, Hülsing, Lee, and Wu. *Fixing and Mechanizing the Security Proof of Fiat-Shamir with Aborts and Dilithium*. CRYPTO 2023
PQC security proofs in EasyCrypt


Hülsing, Meijers, and Strub. *Formal Verification of Saber’s Public-Key Encryption Scheme in EasyCrypt*. CRYPTO 2022

Barbosa, Barthe, Doczkal, Don, Fehr, Grégoire, Huang, Hülsing, Lee, and Wu. *Fixing and Mechanizing the Security Proof of Fiat-Shamir with Aborts and Dilithium*. CRYPTO 2023

3 properties for crypto software

1. Efficiency
   - Speed (clock cycles)
   - RAM usage
   - Binary size
   - Energy consumption

2. Security
   - Don’t leak secrets
   - “Constant-time”
   - Resist Spectre attacks
   - Resist Power/EM attacks
   - Fault protection
   - Easy-to-use APIs

3. Correctness
   - Functionally correct
   - Memory safety
   - Thread safety
   - Termination
• Language with “C-like” syntax
• Programming in Jasmin is much closer to assembly:
  • Generally: 1 line in Jasmin $\rightarrow$ 1 line in assembly
  • A few exceptions, but highly predictable
  • Compiler does not schedule code
  • Compiler does not spill registers
• Language with “C-like” syntax
• Programming in Jasmin is much closer to assembly:
  • Generally: 1 line in Jasmin → 1 line in assembly
  • A few exceptions, but highly predictable
  • Compiler does not schedule code
  • Compiler does not spill registers
• Compiler is formally proven to preserve semantics
• Compiler is formally proven to preserve constant-time property\(^4\)

Jasmin – assembly in your head

• Language with “C-like” syntax
• Programming in Jasmin is much closer to assembly:
  • Generally: 1 line in Jasmin $\rightarrow$ 1 line in assembly
  • A few exceptions, but highly predictable
  • Compiler does not schedule code
  • Compiler does not spill registers
• Compiler is formally proven to preserve semantics
• Compiler is formally proven to preserve constant-time property\(^4\)
• Many new features since 2017 paper!

---

\(^4\)Barthe, Grégoire, Laporte, and Priya. *Structured Leakage and Applications to Cryptographic Constant-Time and Cost.* ACM CCS 2022
C code

```c
#include <stdio.h>

int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

Jasmin code

• We don't implement main in Jasmin
• We don't have I/O in Jasmin (yet)
C code

```c
#include <stdio.h>

int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

Jasmin code

- We don’t implement main in Jasmin
- We don’t have I/O in Jasmin (yet)
Jasmin “Hello World!”

```jasmin
export fn add42(reg u64 x) -> reg u64 {
    reg u64 r;
    r = x;
    r += 42;
    return r;
}
```
https://cryptojedi.org/programming/jasmin.shtml
Jasmin “Hello World!”

```jasmin
param int VLEN = 128;

fn addvec_for(reg ptr u32[VLEN] r a b) -> stack u32[VLEN]
{
    inline int i;
    reg u32 t;

    for i = 0 to VLEN {
        t = a[i];
        t += b[i];
        r[i] = t;
    }
    return r;
}
```
param int VLEN = 128;

fn addvec_while(reg ptr u32[VLEN] r a b) -> stack u32[VLEN]
{
    reg u64 i;
    reg u32 t;
    i = 0;
    while (i < VLEN) {
        t = a[(int)i];
        t += b[(int)i];
        r[(int)i] = t;
        i += 1;
    }
    return r;
}
param int VLEN = 128;

fn addvec_avx2(reg ptr u32[VLEN] r a b) -> stack u32[VLEN]
{
    inline int i;
    reg u256 t0, t1;

    for i = 0 to VLEN/8 {
        t0 = a.[u256 (int)(32 *64u i)];
        t1 = b.[u256 (int)(32 *64u i)];
        t0 = #VPADD_8u32(t0, t1);
        r.[u256 (int)(32 *64u i)] = t0;
    }
    return r;
}
Efficiency of Jasmin code

- Can do (almost) everything you can do in assembly
  - (Almost) full control
  - Architecture-specific implementations

- Easier to write and maintain than assembly:
  - Loops, conditionals
  - Function calls (optional: inline)
  - Function-local variables
  - Register and stack arrays
  - Register and stack allocation

- No raw pointers, no pointer arithmetic
- Very limited control over register allocation
- As efficient as hand-optimized assembly!
Efficiency of Jasmin code

- Can do (almost) everything you can do in assembly
  - (Almost) full control
  - Architecture-specific implementations
- Easier to write and maintain than assembly:
  - Loops, conditionals
  - Function calls (optional: inline)
  - Function-local variables
  - Register and stack arrays
  - Register and stack allocation
Efficiency of Jasmin code

- Can do (almost) everything you can do in assembly
  - (Almost) full control
  - Architecture-specific implementations

- Easier to write and maintain than assembly:
  - Loops, conditionals
  - Function calls (optional: inline)
  - Function-local variables
  - Register and stack arrays
  - Register and stack allocation

- No raw pointers, no pointer arithmetic

- Very limited control over register allocation
Efficiency of Jasmin code

- Can do (almost) everything you can do in assembly
  - (Almost) full control
  - Architecture-specific implementations
- Easier to write and maintain than assembly:
  - Loops, conditionals
  - Function calls (optional: inline)
  - Function-local variables
  - Register and stack arrays
  - Register and stack allocation
- No raw pointers, no pointer arithmetic
- Very limited control over register allocation

As efficient as hand-optimized assembly!
Security – “constant time”

- Enforce constant-time on Jasmin source level
- Every piece of data is either secret or public
- Flow of secret information is traced by type system

“Any operation with a secret input produces a secret output”
Security – “constant time”

- Enforce constant-time on Jasmin source level
- Every piece of data is either secret or public
- Flow of secret information is traced by type system
  
  “Any operation with a secret input produces a secret output”

- Branch conditions and memory indices need to be public
Security – “constant time”

- Enforce constant-time on Jasmin source level
- Every piece of data is either secret or public
- Flow of secret information is traced by type system
  
  “Any operation with a secret input produces a secret output”

- Branch conditions and memory indices need to be public
- In principle can do this also in, e.g., Rust (secret_integers crate)
- **Remember:** Jasmin compiler is verified to preserve constant-time!
Security – “constant time”

- Enforce constant-time on Jasmin source level
- Every piece of data is either secret or public
- Flow of secret information is traced by type system

"Any operation with a secret input produces a secret output"

- Branch conditions and memory indices need to be public
- In principle can do this also in, e.g., Rust (secret_integers crate)

- **Remember:** Jasmin compiler is verified to preserve constant-time!
- Explicit #declassify primitive to move from secret to public
- #declassify creates a proof obligation!
Spectre v1 ("Speculative bounds-check bypass")

```
stack u8[10] public;
stack u8[32] secret;
reg u8 t;
reg u64 r, i;

i = 0;
while(i < 10) {
    t = public[(int) i] ;
    r = leak(t);
    ...
}
```
    reg u64 rkoffset;
    state = in;

    state ^= rkeys[0];
    rkoffset = 0;
    while(rkoffset < 9*16) {
        rk = rkeys.[(int)rkoffset];
        state = #AESENC(state, rk);
        rkoffset += 16;
    }
    rk = rkeys[10];
    #declassify state = #AESENCLAST(state, rk);
    return state;
}
It’s more subtle than this

Spectre declassified

- Caller is free to leak (declassified) state
- Very common in crypto: ciphertext is actually sent!
- state is not “out of bounds” data, it’s “early data”
- Must not speculatively #declassify early!

Fencing

- Can prevent speculation through barriers (LFENCE)
- Protecting all branches is possible but costly
Countermeasures

**Fencing**
- Can prevent speculation through **barriers** (LFENCE)
- Protecting *all* branches is possible but costly

**Speculative Load Hardening**
- Idea: maintain misprediction predicate $m_s$ (in a register)
- At every branch use arithmetic to update predicate
- Option 1: Mask every loaded value with $m_s$
- Option 2: Mask every address with $m_s$
- Effect: during misspeculation “leak” constant value
Countermeasures

Fencing

• Can prevent speculation through barriers (LFENCE)
• Protecting all branches is possible but costly

Speculative Load Hardening

• Idea: maintain misprediction predicate ms (in a register)
• At every branch use arithmetic to update predicate
• Option 1: Mask every loaded value with ms
• Option 2: Mask every address with ms
• Effect: during misspeculation “leak” constant value
• Implemented in LLVM since version 8
  • Still large performance overhead
  • No formal guarantees of security
Selective SLH

Do we need to mask/protect all loads?

- No need to mask loads into registers that never enter leaking instructions.
- Secret registers never enter leaking instructions!
- Obvious idea: mask only loads into public registers.
Selective SLH

Do we need to mask/protect all loads?

- No need to mask loads into registers that never enter leaking instructions
Do we need to mask/protect all loads?

- No need to mask loads into registers that never enter leaking instructions
- secret registers never enter leaking instructions!
- Obvious idea: mask only loads into public registers
Extending the type system

- Type system gets three security levels:
  - secret: secret
  - public: public, also during misspeculation
  - transient: public, but possibly secret during misspeculation

Maintain misspeculation flag $ms$:
- $ms = #init_msf()$: Translate to LFENCE, set $ms$ to zero
- $ms = #set_msf(b, ms)$: Set $ms$ according to branch condition $b$

Branches invalidate $ms$

Two operations to lower level:
- $x = #protect(x, ms)$: Go from transient to public
- $#protect$ translates to mask by $ms$
- $#declassify r$: Go from secret to transient
- $#declassify$ requires cryptographic proof/argument

Still: allow branches and indexing only for public
Extending the type system

- Type system gets three security levels:
  - secret: secret
  - public: public, also during misspeculation
  - transient: public, but possibly secret during misspeculation

- Maintain misspeculation flag ms:
  - ms = #init_msf(): Translate to LFENCE, set ms to zero
  - ms = #set_msf(b, ms): Set ms according to branch condition b
  - Branches invalidate ms
Extending the type system

- Type system gets three security levels:
  - secret: secret
  - public: public, also during misspeculation
  - transient: public, but possibly secret during misspeculation

- Maintain misspeculation flag ms:
  - ms = init_msf(): Translate to LFENCE, set ms to zero
  - ms = set_msf(b, ms): Set ms according to branch condition b
  - Branches invalidate ms

- Two operations to lower level:
  - x = protect(x, ms): Go from transient to public
  - protect translates to mask by ms
Extending the type system

• Type system gets three security levels:
  • secret: secret
  • public: public, also during misspeculation
  • transient: public, but possibly secret during misspeculation

• Maintain misspeculation flag ms:
  • ms = #init_msf(): Translate to LFENCE, set ms to zero
  • ms = #set_msf(b, ms): Set ms according to branch condition b
  • Branches invalidate ms

• Two operations to lower level:
  • x = #protect(x, ms): Go from transient to public
  • #protect translates to mask by ms
  • #declassify r: Go from secret to transient
  • #declassify requires cryptographic proof/argument
Extending the type system

• Type system gets three security levels:
  • secret: secret
  • public: public, also during misspeculation
  • transient: public, but possibly secret during misspeculation

• Maintain misspeculation flag $ms$:
  • $ms = \#\text{init}_ms()$: Translate to LFENCE, set $ms$ to zero
  • $ms = \#\text{set}_ms(b, ms)$: Set $ms$ according to branch condition $b$
  • Branches invalidate $ms$

• Two operations to lower level:
  • $x = \#\text{protect}(x, ms)$: Go from transient to public
  • $\#\text{protect}$ translates to mask by $ms$
  • $\#\text{declassify} r$: Go from secret to transient
  • $\#\text{declassify}$ requires cryptographic proof/argument

• Still: allow branches and indexing only for public
The special case of crypto

- We know what inputs are **secret** and what inputs are **public**
- Most of the state is actually **secret**
- Most loads do not need **protect**!
The special case of crypto

- We know what inputs are secret and what inputs are public
- Most of the state is actually secret
- Most loads do not need protect!
- Even better: mark additional inputs as secret
- No cost if those inputs don’t flow into leaking instructions
The special case of crypto

- We know what inputs are **secret** and what inputs are **public**
- Most of the state is actually **secret**
- Most loads do not need **protect**!
- Even better: mark additional inputs as **secret**
- No cost if those inputs don’t flow into leaking instructions
- Even better: Spills don’t need **protect** if there is no branch between store and load
The special case of crypto

- We know what inputs are secret and what inputs are public
- Most of the state is actually secret
- Most loads do not need protect!
- Even better: mark additional inputs as secret
- No cost if those inputs don’t flow into leaking instructions
- Even better: Spills don’t need protect if there is no branch between store and load
- Even better: “Spill” public data to MMX registers instead of stack

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Impl.</th>
<th>Op.</th>
<th>CT</th>
<th>SCT</th>
<th>overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChaCha20</td>
<td>avx2</td>
<td>32 B</td>
<td>314</td>
<td>352</td>
<td>12.10</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>32 B xor</td>
<td>314</td>
<td>352</td>
<td>12.10</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>128 B</td>
<td>330</td>
<td>370</td>
<td>12.12</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>128 B xor</td>
<td>338</td>
<td>374</td>
<td>10.65</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>1 KiB</td>
<td>1190</td>
<td>1234</td>
<td>3.70</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>1 KiB xor</td>
<td>1198</td>
<td>1242</td>
<td>3.67</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>1 KiB</td>
<td>18872</td>
<td>18912</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>16 KiB xor</td>
<td>18970</td>
<td>18994</td>
<td>0.13</td>
</tr>
</tbody>
</table>
## Performance impact (Comet Lake cycles)

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Impl.</th>
<th>Op.</th>
<th>CT</th>
<th>SCT</th>
<th>overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly1305</td>
<td>avx2</td>
<td>32 B</td>
<td>46</td>
<td>78</td>
<td>69.57</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>32 B verif</td>
<td>48</td>
<td>84</td>
<td>75.00</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>128 B</td>
<td>136</td>
<td>172</td>
<td>26.47</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>128 B verif</td>
<td>140</td>
<td>170</td>
<td>21.43</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>1 KiB</td>
<td>656</td>
<td>686</td>
<td>4.57</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>1 KiB verif</td>
<td>654</td>
<td>686</td>
<td>4.89</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>16 KiB</td>
<td>8420</td>
<td>8450</td>
<td>0.36</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>16 KiB verif</td>
<td>8416</td>
<td>8466</td>
<td>0.59</td>
</tr>
</tbody>
</table>
## Performance impact (Comet Lake cycles)

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Impl.</th>
<th>Op.</th>
<th>CT</th>
<th>SCT</th>
<th>overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>X25519</td>
<td>mulx</td>
<td>smult</td>
<td>98352</td>
<td>98256</td>
<td>-0.098</td>
</tr>
<tr>
<td></td>
<td>mulx</td>
<td>base</td>
<td>98354</td>
<td>98262</td>
<td>-0.094</td>
</tr>
<tr>
<td>Kyber512</td>
<td>avx2</td>
<td>keypair</td>
<td>25694</td>
<td>25912</td>
<td>0.848</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>enc</td>
<td>35186</td>
<td>35464</td>
<td>0.790</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>dec</td>
<td>27684</td>
<td>27976</td>
<td>1.055</td>
</tr>
<tr>
<td>Kyber768</td>
<td>avx2</td>
<td>keypair</td>
<td>42768</td>
<td>42888</td>
<td>0.281</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>enc</td>
<td>54518</td>
<td>54818</td>
<td>0.550</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>dec</td>
<td>43824</td>
<td>44152</td>
<td>0.748</td>
</tr>
</tbody>
</table>
Correctness – memory and thread safety, termination

- No global variables $\rightarrow$ thread safety
• No global variables → thread safety

• **Static** safety checker:
  • Uses language limitations
  • Ensures termination
  • Ensures memory safety (and prints conditions for inputs)
  • Not part of “standard compilation”: `-checksafety`
Correctness – memory and thread safety, termination

- No global variables → thread safety
- **Static** safety checker:
  - Uses language limitations
  - Ensures termination
  - Ensures memory safety (and prints conditions for inputs)
  - Not part of “standard compilation”: -checksafty

- Some limitations/caveats:
  - Sound, but not complete
  - Very slow (about 1 day for Kyber’s Encaps)
  - Overly strict alignment requirements
  - May need annotations (e.g., #bounded, #no_termination_check)

“I’m carefully optimistic that we have the full proof and optimized software done by summer.”

—me, May 2020
Correctness – the missing link


“I’m carefully optimistic that we have the full proof and optimized software done by summer.”
—me, May 2020

• Started in Feb. 2020 as a “4-month-sabbatical” project
“I’m carefully optimistic that we have the full proof and optimized software done by summer.”
—me, May 2020

- Started in Feb. 2020 as a “4-month-sabbatical” project
- 3-year effort, 12 authors (so far)
- A lot of work to link Jasmin implementation with EasyCrypt specification
- This is per-implementation effort, not per-scheme effort
More proof automation!

- Integrate with CryptoLine (https://github.com/fmlab-iis/cryptoline)$^5$
  - (semi-)automated proof of branch-free arithmetic
  - “Prove without understanding code”
- Automated equivalence proving...
More proof automation!

- Integrate with CryptoLine ([https://github.com/fmlab-iis/cryptoline](https://github.com/fmlab-iis/cryptoline))
  - (semi-)automated proof of branch-free arithmetic
  - “Prove without understanding code”
- Automated equivalence proving...

Beyond Spectre v1

- Spectre v2: Avoid by not using indirect branches
- Spectre protection requires separation of crypto code!

---

Support more architectures

- 32-bit Arm (ARMv7ME): works, needs users!
- Opentitan’s OTBN: work in progress
- 64-bit ARM and RISC-V: very early WIP
Support more architectures

- 32-bit Arm (ARMv7ME): works, needs users!
- Opentitan’s OTBN: work in progress
- 64-bit ARM and RISC-V: very early WIP

Secure interfacing

- Currently use C function-call ABI (caller/callee contract through documentation)
- Check/Enforce caller requirements?
- Stronger safety notions (e.g., interfacing with Rust)
The big challenge

Make high-assurance tools mainstream/default!
Join the effort:

https://formosa-crypto.org

Use the results:

https://github.com/formosa-crypto/libjade