Formosa Crypto – high-assurance crypto software in practice

Peter Schwabe
February 20, 2024
Cryptographic software

- Primitives, no protocols
- “Secure-channel” primitives
The setting for this talk

Cryptographic software

- Primitives, no protocols
- “Secure-channel” primitives
- Only software-visible side channels
The setting for this talk

Cryptographic 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)
- 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!
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
Post-quantum crypto

- 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 . . . ?
Post-quantum crypto

- 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 NISTPQC KEM Lattice Finalists: Kyber, SABER, and NTRU.**
Advanced microarchitectural side channels

- MELTDOWN
- CACHE OUT
- Hertzbleed
Tools that aren’t built for crypto

“...implementations shall consist of source code written in ANSI C.”

—NIST PQC Call for Proposals, 2017

- No memory safety
- Finicky semantics
  - Undefined behavior
  - Implementation-specific behavior
  - Context-specific behavior
- No mandatory initialization
- No (optional) runtime checks
Tools that aren’t built for crypto

“…implementations shall consist of source code written in ANSI C.”

—NIST PQC Call for Proposals, 2017

but... Rust!

- No memory safety
- Finicky semantics
  - Undefined behavior
  - Implementation-specific behavior
  - Context-specific behavior
- No mandatory initialization
- No (optional) runtime checks
- Memory safe
- More clear semantics (?)
- Mandatory variable initialization
- (Optional) runtime checks for, e.g., overflows
Tools that aren’t built for crypto

Lack of security features

- No concept of secret vs. public data
- No preservation of “constant-time”
- Limited protection against microarchitectural attacks
- Limited support for erasure of sensitive data
Let’s fix those tools!

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

—Simon, Chisnall, Anderson, 2018
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/rfc Pulse pull/2859
• Implementation in LLVM is massive effort, no real progress, yet
Spectre protections in LLVM

- Carruth, 2019: Spectre v1 countermeasure in LLVM\(^1\) (see later in the talk)
- “does not defend against secret data already loaded from memory and residing in registers”

\(^1\)https://llvm.org/docs/SpeculativeLoadHardening.html

\(^2\)Ultimate SLH: Taking Speculative Load Hardening to the Next Level. USENIX Security, 2023
Spectre protections in LLVM

- Carruth, 2019: Spectre v1 countermeasure in LLVM\(^1\) (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\(^2\)
- Report and proposed patches to LLVM in March 2022
- September 2022: **Status: WontFix (was: New)**

---

\(^1\)https://llvm.org/docs/SpeculativeLoadHardening.html

\(^2\)Ultimate SLH: Taking Speculative Load Hardening to the Next Level. USENIX Security, 2023
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 $>200$ people
The toolchain and workflow
The toolchain and workflow
The toolchain and workflow
Running example: Kyber

“The public-key encryption and key-establishment algorithm that will be standardized is CRYSTALS-KYBER.”

—NIST IR 8413-upd1

- Lattice-based KEM, joint work with Avanzi, Bos, Ding, Ducas, Kiltz, Lepoint, Lyubashevsky, Schanck, Schwabe, Seiler, and Stehlé.
- Three parameter sets; “recommended” is Kyber768
- FIPS draft standard public for comments: https://csrc.nist.gov/pubs/fips/203/ipd
- Already deployed in TLS by Google and Cloudflare
Functional correctness of Kyber implementations

Functional correctness of Kyber implementations

Functional correctness of Kyber implementations
Implementing in Jasmin

- 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

---

Implementing in Jasmin

- 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
- Static (trusted) safety checker
- Compiler is formally proven to preserve constant-time property\(^3\)

\(^3\)Barthe, Grégoire, Laporte, and Priya. \textit{Structured Leakage and Applications to Cryptographic Constant-Time and Cost}. ACM CCS 2022
Efficiency of Jasmin code

- Can do (almost) everything you can do in assembly
- Architecture-specific implementations
- Small limitations to enable static safety checking (no raw pointers)
Efficiency of Jasmin code

• Can do (almost) everything you can do in assembly
• Architecture-specific implementations
• Small limitations to enable static safety checking (no raw pointers)
• 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

### Performance of Kyber code

<table>
<thead>
<tr>
<th>Implementation</th>
<th>operation</th>
<th>Skylake</th>
<th>Haswell</th>
<th>Comet Lake</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/asm AVX2</td>
<td>keygen</td>
<td>49572</td>
<td>47280</td>
<td>41682</td>
</tr>
<tr>
<td></td>
<td>encaps</td>
<td>60018</td>
<td>62900</td>
<td>55956</td>
</tr>
<tr>
<td></td>
<td>decaps</td>
<td>45854</td>
<td>47784</td>
<td>43906</td>
</tr>
<tr>
<td>Jasmin AVX2 (fully verified)</td>
<td>keygen</td>
<td>106578</td>
<td>96296</td>
<td>93244</td>
</tr>
<tr>
<td></td>
<td>encaps</td>
<td>119308</td>
<td>111536</td>
<td>107474</td>
</tr>
<tr>
<td></td>
<td>decaps</td>
<td>105336</td>
<td>98328</td>
<td>96564</td>
</tr>
<tr>
<td>Jasmin AVX2 (fully optimized)</td>
<td>keygen</td>
<td>50004</td>
<td>48800</td>
<td>45046</td>
</tr>
<tr>
<td></td>
<td>encaps</td>
<td>65132</td>
<td>63988</td>
<td>59496</td>
</tr>
<tr>
<td></td>
<td>decaps</td>
<td>50340</td>
<td>51444</td>
<td>48172</td>
</tr>
</tbody>
</table>
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!
• 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
```c
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);
    ...
}
```
Extending the type system

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

Don't branch or index memory based on secret or transient data

Guide programmer to protect code

Selective speculative load hardening (selSLH):
- Misspeculation flag in register
- Mask "transient" values with flag before leaking them

Overhead for Kyber768 (on Intel Comet Lake):
- 0.28% for Keypair
- 0.55% for Encaps
- 0.75% for Decaps

Exploits synergies with protections against "traditional" timing attacks

Ammanaghatta Shivakumar, Barthe, Grégoire, Laporte, Oliveira, Priya, Schwabe, and Tabary-Maujean.
Typing High-Speed Cryptography against Spectre v1.
Extending the type system

- Type system gets three security levels:
  - secret: secret
  - public: public, also during misspeculation
  - transient: public, but possibly secret during misspeculation
- Don't branch or index memory based on secret or transient data
Extending the type system

- Type system gets three security levels:
  - secret: secret
  - public: public, also during misspeculation
  - transient: public, but possibly secret during misspeculation
- Don’t branch or index memory based on secret or transient data
- Guide programmer to protect code
- Selective speculative load hardening (selSLH):
  - Misspeculation flag in register
  - Mask “transient” values with flag before leaking them
Extending the type system

- Type system gets three security levels:
  - secret: secret
  - public: public, also during misspeculation
  - transient: public, but possibly secret during misspeculation
- Don’t branch or index memory based on secret or transient data
- Guide programmer to protect code
- Selective speculative load hardening (selSLH):
  - Misspeculation flag in register
  - Mask “transient” values with flag before leaking them
- Overhead for Kyber768 (on Intel Comet Lake):
  - 0.28% for Keypair
  - 0.55% for Encaps
  - 0.75% for Decaps
- Exploits synergies with protections against “traditional” timing attacks

“...A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys”

—FIPS 140-3, Section 9.7.A
“...A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys”

—FIPS 140-3, Section 9.7.A

Goal of zeroization

Scrub all (sensitive) data from memory (stack) and registers when crypto routine returns.
“...A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys”

—FIPS 140-3, Section 9.7.A

**Goal of zeroization**

Scrub all (sensitive) data from memory (stack) and registers when crypto routine returns.

**Failure modes**

0. Don't perform any zeroization
“…A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys”

—FIPS 140-3, Section 9.7.A

Goal of zeroization

Scrub all (sensitive) data from memory (stack) and registers when crypto routine returns.

Failure modes

0. Don’t perform any zeroization

1. Dead-store elimination
Security – zeroization

“…A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys”

—FIPS 140-3, Section 9.7.A

Goal of zeroization

Scrub all (sensitive) data from memory (stack) and registers when crypto routine returns.

Failure modes

0. Don’t perform any zeroization
1. Dead-store elimination
2. Only API-level stack zeroization
“...A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys”

—FIPS 140-3, Section 9.7.A

Goal of zeroization

Scrub all (sensitive) data from memory (stack) and registers when crypto routine returns.

Failure modes

0. Don't perform any zeroization
1. Dead-store elimination
2. Only API-level stack zeroization
3. Don’t scrub source-level invisible data
“... A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys”

—FIPS 140-3, Section 9.7.A

Goal of zeroization

Scrub all (sensitive) data from memory (stack) and registers when crypto routine returns.

Failure modes

0. Don’t perform any zeroization
1. Dead-store elimination
2. Only API-level stack zeroization
3. Don’t scrub source-level invisible data
4. Mis-estimate stack space when scrubbing from caller
Solution in Jasmin compiler

Zeroize used stack space and registers when returning from export function

Solution in Jasmin compiler

Zeroize used stack space and registers when returning from export function

- Make use of multiple features of Jasmin:
  - Compiler has global view
  - All stack usage is known at compile time
  - Entry/return point is clearly defined

Security – zeroization (ctd.)

Solution in Jasmin compiler

Zeroize used stack space and registers when returning from export function

● Make use of multiple features of Jasmin:
  ● Compiler has global view
  ● All stack usage is known at compile time
  ● Entry/return point is clearly defined

● Performance overhead for Kyber768:
  ● 0.59% for Keypair
  ● 0.24% for Encaps
  ● 1.04% for Decaps

Libjade – the interface to Formosa Crypto

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

- Collection of primitive implementations rather than library
- “A library to be used by libraries”
Libjade – the interface to Formosa Crypto

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

- Collection of primitive implementations rather than library
- “A library to be used by libraries”
- Example:
  
  ```
  cd src/crypto_kem/kyber/kyber768/amd64/ref/ && make
  ```
  
  will build
  
  ```
  src/crypto_kem/kyber/kyber768/amd64/ref/kem.s
  ```
  
  with API described in
  
  ```
  src/crypto_kem/kyber/kyber768/amd64/ref/include/api.h
  ```
• Releases contain
  • compiled assembly files + headers
  • jasmin files
  • usage examples written in C
• Latest release: 2023.05-1
• Releases contain
  • compiled assembly files + headers
  • jasmin files
  • usage examples written in C

• Latest release: 2023.05-1

• Plans for next release:
  • Integrate EasyCrypt proofs (covered by CI)
  • Integrate/consolidate various features
  • Special focus on Kyber-768
Challenges, ongoing work, TODOs

More proof automation!

• Integrate with CryptoLine (https://github.com/fmlab-iis/cryptoline)\textsuperscript{4}
  • (semi-)automated proof of branch-free arithmetic
  • “Prove without understanding code”
• Automated equivalence proving...

\textsuperscript{4}Fu, Liu, Shi, Tsai, Wang, and Yang. Signed Cryptographic Program Verification with Typed CryptoLine. ACM CCS 2019
More proof automation!

- Integrate with CryptoLine (https://github.com/fmlab-iis/cryptoline)\(^4\)
  - (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 v4: Use SSBD: https://github.com/tyhicks/ssbd-tools
- Spectre protection requires separation of crypto code!

\(^4\)Fu, Liu, Shi, Tsai, Wang, and Yang. Signed Cryptographic Program Verification with Typed CryptoLine. ACM CCS 2019
Support more architectures

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

- 32-bit Arm (ARMv7ME): works, still “experimental”
- 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!
https://formosa-crypto.org