MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

# Formosa Crypto – high-assurance crypto software in practice

Peter Schwabe

November 15, 2023

# 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)

# 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!

# PQC

- More assumptions, more schemes, more parameters, **more code**
- More complexity in implementations, protocols, and proofs

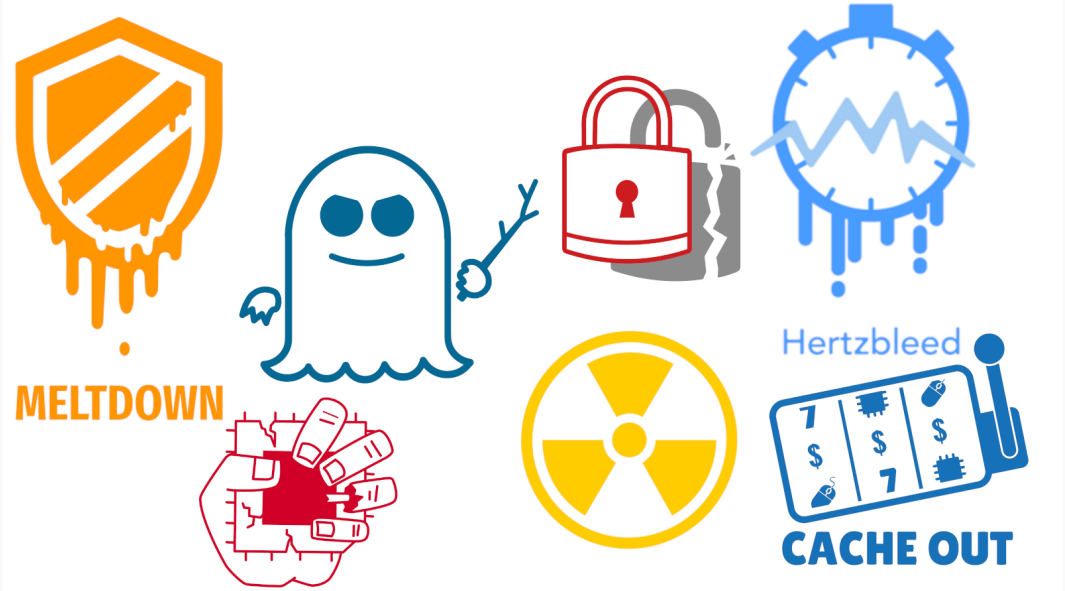
# PQC

- More assumptions, more schemes, more parameters, **more code**
- More complexity in implementations, protocols, and proofs
- Many early 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 . . . ?

# PQC

- More assumptions, more schemes, more parameters, **more code**
- More complexity in implementations, protocols, and proofs
- Many early 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



# 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

- 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

# 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

# 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 ≈ 30–40 people
- Discussion forum with >180 people

University of BRISTOL

THE UNIVERSITY OF MELBOURNE

CASA
CYBER SECURITY IN THE AGE OF LARGE-SCALE ADVERSARIES

MAX PLANCK INSTITUTE FOR SECURITY AND PRIVACY

institute IMDEA software

U.PORTO

Radboud University

INESCTEC

RUHR UNIVERSITÄT BOCHUM RUB

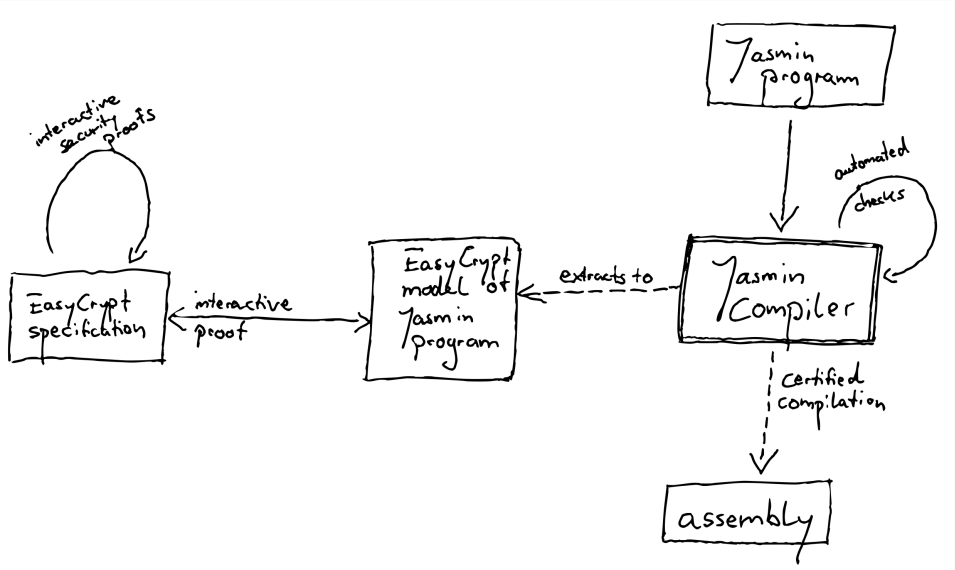
Inria INVENTEURS DU MONDE NUMÉRIQUE

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

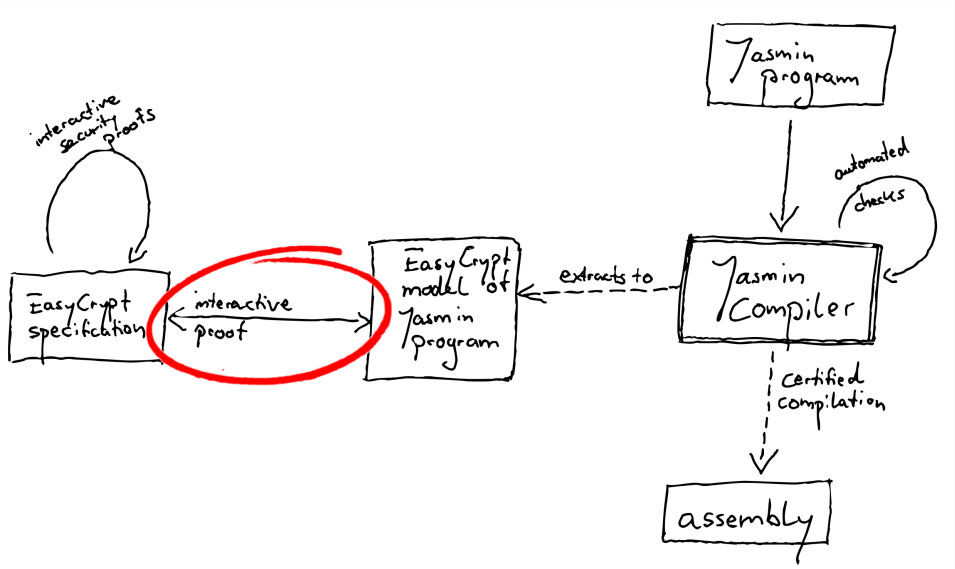
# Joint work with. . .

Aaron Kaiser, Adrien Koutsos, Alley Stoughton, Amber Sprenkels, Andreas Hülsing, Antoine Séré, Basavesh Ammanaghatta Shivakumar, **Benjamin Grégoire**, Benjamin Lipp, Bo-Yin Yang, Bow-Yaw Wang, Chitchanok Chuengsatiansup, Christian Doczkal, Daniel Genkin, Denis Firsov, Fabio Campos, François Dupressoir, Gilles Barthe, Hugo Pacheco, Jack Barnes, **Jean-Christophe Léchenet**, José Bacelar Almeida, Kai-Chun Ning, Lionel Blatter, Lucas Tabary-Maujean, Manuel Barbosa, Matthias Meijers, Miguel Quaresma, Ming-Hsien Tsai, Peter Schwabe, Pierre Boutry, Pierre-Yves Strub, Ruben Gonzalez, Rui Qi Sim, Sabrina Manickam, **Santiago Arranz Olmos**, Sioli O'Connell, Sunjay Cauligi, Swarn Priya, Tiago Oliveira, Vincent Hwang, **Vincent Laporte**, William Wang, Yi Lee, Yuval Yarom, Zhiyuan Zhang

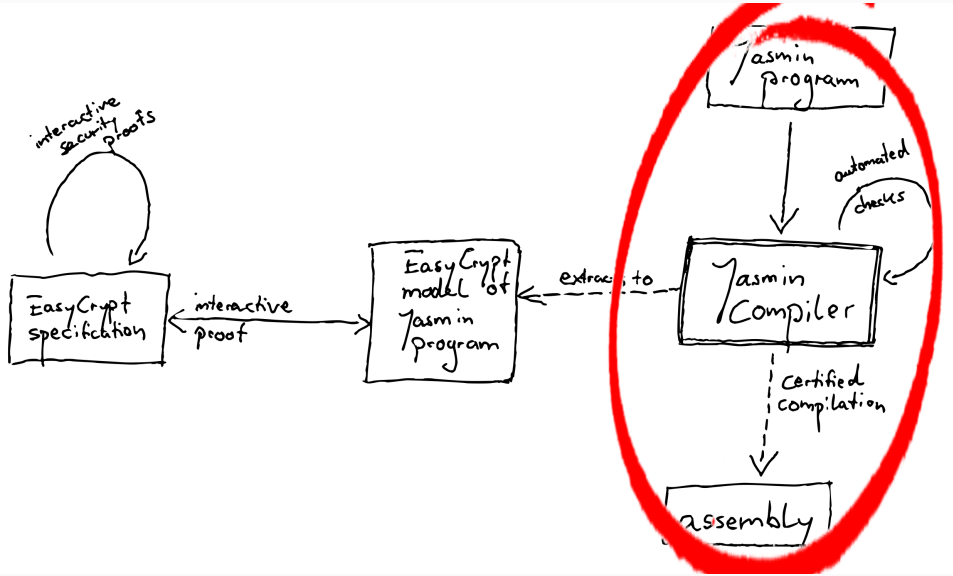
# The toolchain and workflow



# The toolchain and workflow



# The toolchain and workflow



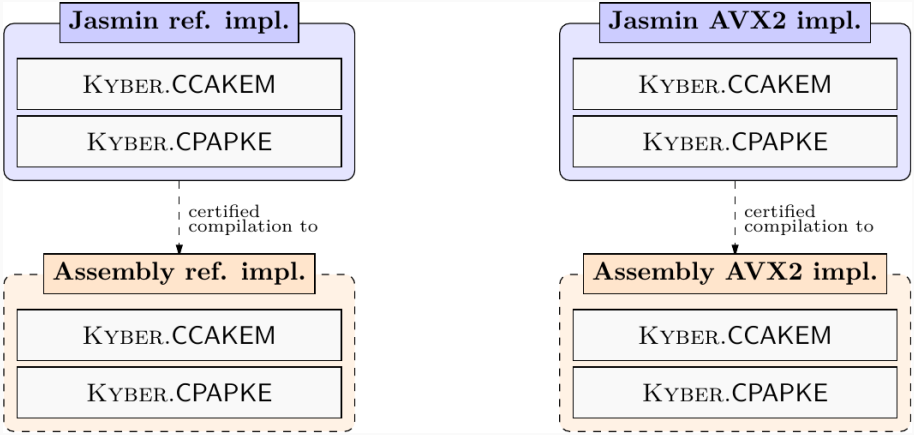
# Running example: Kyber

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

—NIST IR 8413-upd1

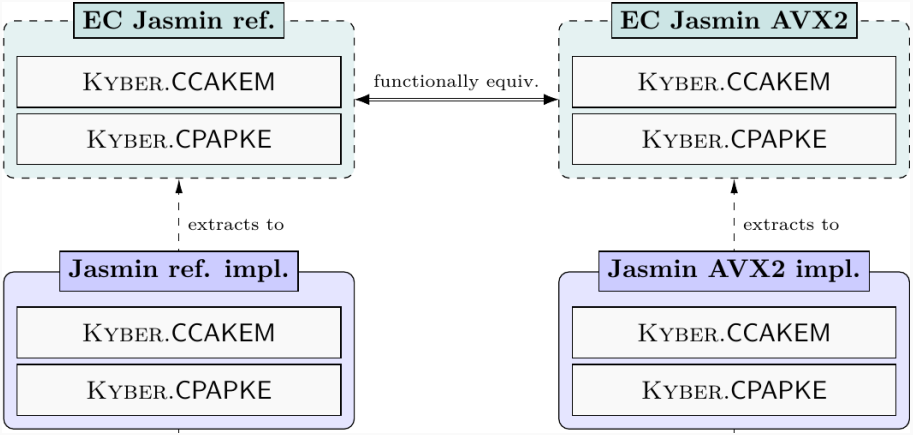
- Lattice-based KEM, joint work with Avanzi, Bos, Ding, Ducas, Kiltz, Lepoint, Lyubashevsky, Schanck, Schwabe, Seiler, and Stehlé.
- Three parameter sets; "recommended" is Kyber768
- FIPS draft standard public for comments: `https://csrc.nist.gov/pubs/fips/203/ipd`
- Already deployed in TLS by Google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

# Functional correctness of Kyber implementations



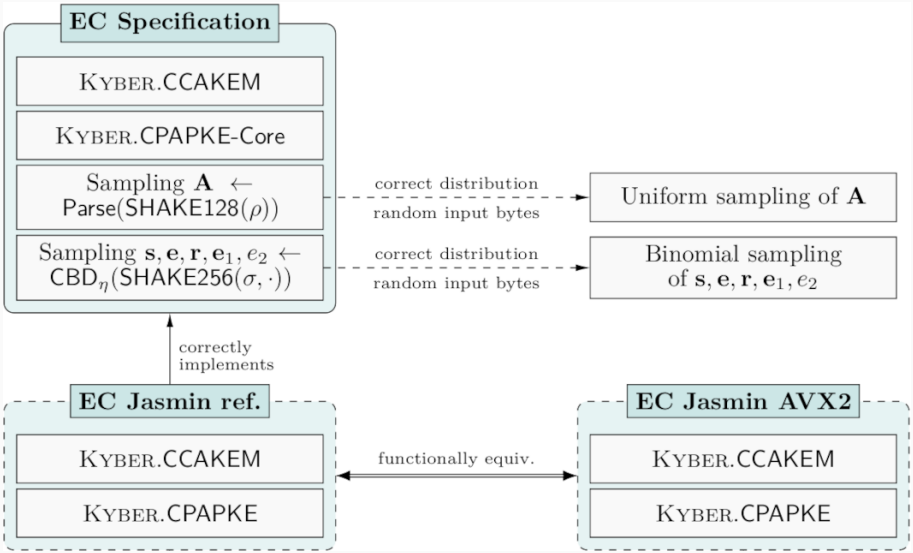

Almeida, Barbosa, Barthe, Grégoire, Laporte, Léchenet, Oliveira, Pacheco, Quaresma, Schwabe, Séré, and Strub. *Formally verifying Kyber – Episode IV: Implementation Correctness.* TCHES 2023-3.

# Functional correctness of Kyber implementations




Almeida, Barbosa, Barthe, Grégoire, Laporte, Léchenet, Oliveira, Pacheco, Quaresma, Schwabe, Séré, and Strub. *Formally verifying Kyber – Episode IV: Implementation Correctness.* TCHES 2023-3.

# Functional correctness of Kyber implementations



# Implementing in Jasmin

Almeida, Barbosa, Barthe, Blot, Grégoire, Laporte, Oliveira, Pacheco, Schmidt, Strub. *Jasmin: High-Assurance and High-Speed Cryptography.* ACM CCS 2017

- 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

[1] Barthe, Grégoire, Laporte, and Priya. *Structured Leakage and Applications to Cryptographic Constant-Time and Cost.* ACM CCS 2022

# Implementing in Jasmin

Almeida, Barbosa, Barthe, Blot, Grégoire, Laporte, Oliveira, Pacheco, Schmidt, Strub. *Jasmin: High-Assurance and High-Speed Cryptography.* ACM CCS 2017

- 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[1]

[1] Barthe, Grégoire, Laporte, and Priya. *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

| Implementation | operation | Skylake | Haswell | Comet Lake |
|---|---|---|---|---|
| C/asm AVX2 | keygen | 49572 | 47280 | 41682 |
| | encaps | 60018 | 62900 | 55956 |
| | decaps | 45854 | 47784 | 43906 |
| Jasmin AVX2 (fully verified) | keygen | 106578 | 96296 | 93244 |
| | encaps | 119308 | 111536 | 107474 |
| | decaps | 105336 | 98328 | 96564 |
| Jasmin AVX2 (fully optimized) | keygen | 50004 | 48800 | 45046 |
| | encaps | 65132 | 63988 | 59496 |
| | decaps | 50340 | 51444 | 48172 |

# 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`

# Security – 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);
  ...
}
```

# Extending the type system

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

# 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

Ammanaghatta Shivakumar, Barthe, Grégoire, Laporte, Oliveira, Priya, Schwabe, and Tabary-Maujean. *Typing High-Speed Cryptography against Spectre v1.* IEEE S&P 2023.

# Security – zeroization

*"...A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys"*

—FIPS 140-3, Section 9.7.A

# 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.

# 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

# 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

# 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

# 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
3. Don't scrub source-level invisible data

# 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
3. Don't scrub source-level invisible data
4. Mis-estimate stack space when scrubbing from caller

# Security – zeroization (ctd.)

## Solution in Jasmin compiler

Zeroize used stack space and registers when returning from export function

Arranz Olmos, Barthe, Gonzalez, Grégoire, Laporte, Léchenet, Oliveira, Schwabe: *High-assurance zeroization.* TCHES 2024-1 (to appear).

# 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

Arranz Olmos, Barthe, Gonzalez, Grégoire, Laporte, Léchenet, Oliveira, Schwabe: *High-assurance zeroization.* TCHES 2024-1 (to appear).

# 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

Arranz Olmos, Barthe, Gonzalez, Grégoire, Laporte, Léchenet, Oliveira, Schwabe: *High-assurance zeroization.* TCHES 2024-1 (to appear).

# Challenges, ongoing work, TODOs

## More proof automation!

- Integrate with CryptoLine (`https://github.com/fmlab-iis/cryptoline`)[2]
  - (semi-)automated proof of branch-free arithmetic
  - "Prove without understanding code"
- Automated equivalence proving. . .

[2] Fu, Liu, Shi, Tsai, Wang, and Yang. Signed Cryptographic Program Verification with Typed CryptoLine. ACM CCS 2019

# Challenges, ongoing work, TODOs

## More proof automation!

- Integrate with CryptoLine (`https://github.com/fmlab-iis/cryptoline`)[2]
  - (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!**

[2] Fu, Liu, Shi, Tsai, Wang, and Yang. Signed Cryptographic Program Verification with Typed CryptoLine. ACM CCS 2019

# Challenges, ongoing work, TODOs, part 2

## 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

# Challenges, ongoing work, TODOs, part 2

## 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!

# Resources

Join the effort:

https://formosa-crypto.org

Use the results:

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