MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

Formosa Crypto: high-assurance, high-security crypto software

Peter Schwabe

December 5, 2024

The setting for this talk

Cryptographic software

- Primitives, no protocols
- “Secure-channel” primitives
- Post-quantum primitives

The setting for this talk

Cryptographic software

- Primitives, no protocols
- "Secure-channel" primitives
- Post-quantum primitives
- Only software-visible side channels

The setting for this talk

Cryptographic software

- Primitives, no protocols
- "Secure-channel" primitives
- Post-quantum primitives
- Only software-visible side channels
- Big CPUs

Running example: ML-KEM-768

"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.
- Standardized in August 2024 as ML-KEM in FIPS 203
- Three parameter sets; "recommended" is **ML-KEM-768**
  https://csrc.nist.gov/pubs/fips/203/final
- Already deployed in TLS by Google and Cloudflare
- Also already used in, e.g., Signal and iMessage

The one-slide summary of ML-KEM

Lattice-based encryption K-PKE

- Arithmetic in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $q = 3329$, $n = 256$
- Computations of the form $As + e$ with $A \in \mathcal{R}_q^{3\times 3}$ and $s, e \in \mathcal{R}_q^3$
- Security reduction from variant of Module-Learning-with-Errors (MLWE)

The one-slide summary of ML-KEM

Lattice-based encryption K-PKE

- Arithmetic in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $q = 3329$, $n = 256$
- Computations of the form $As + e$ with $A \in \mathcal{R}_q^{3\times 3}$ and $s, e \in \mathcal{R}_q^3$
- Security reduction from variant of Module-Learning-with-Errors (MLWE)

Fujisaki-Okamoto Transform

- Required to achieve active (IND-CCA) security
- Enforce honestly generated ciphertexts
- Encapsulation generates all randomness as PRF(H(m))
- Decapsulation re-encrypts and compares ciphertexts

How do we implement primitives such as ML-KEM?

"... 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

How do we implement primitives such as ML-KEM?

"... 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

How do we implement primitives such as ML-KEM?

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/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[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

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 >280 people

University of BRISTOL

UNIVERSITÄT KLAGENFURT

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

Formosan black bear

24 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

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]

Formosan black bear

Conservation status

Extinct | Threatened | Least Concern

EX EW CR EN VU NT LC

Vulnerable (IUCN 3.1)[1]

Physical characteristics [edit]

The V-shaped white mark on a bear's chest

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

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 >280 people

University of BRISTOL

UNIVERSITÄT KLAGENFURT

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

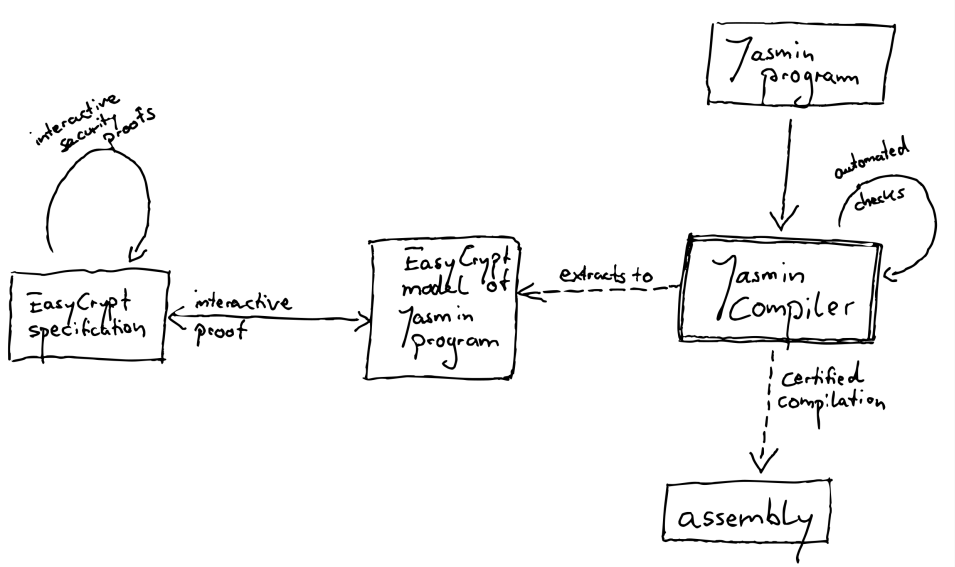
Inria INVENTEURS DU MONDE NUMÉRIQUE

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

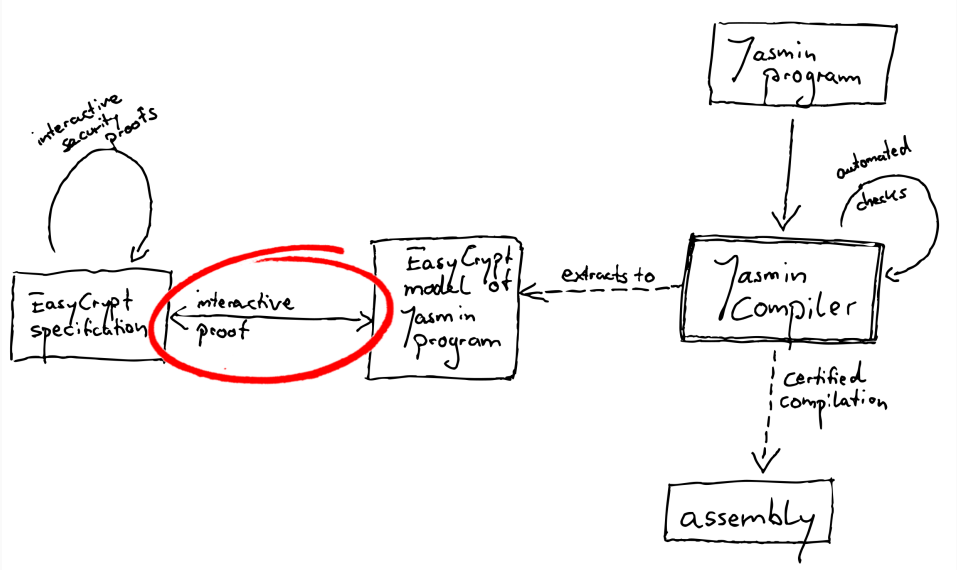
Joint work with...

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, Cameron Low, 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

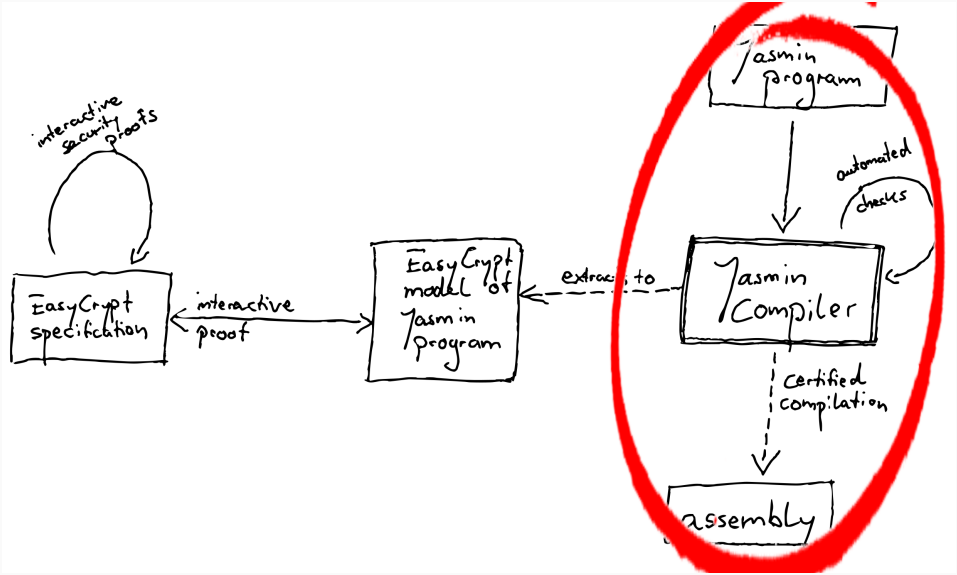
The toolchain and workflow



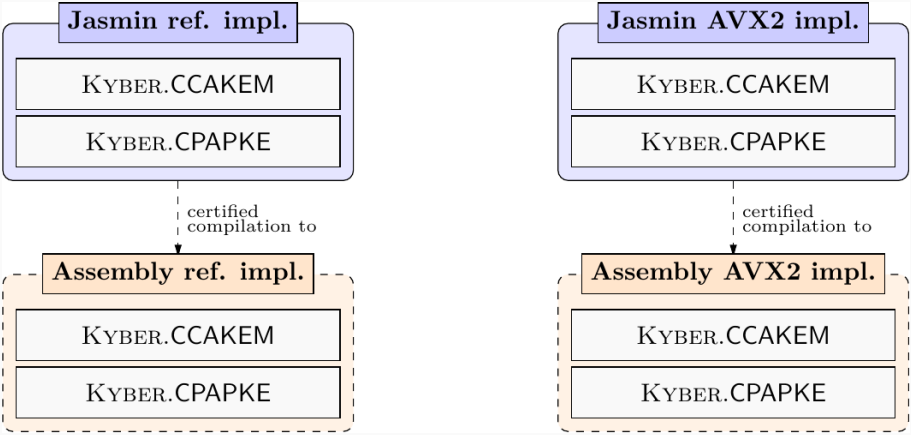
The toolchain and workflow



The toolchain and workflow

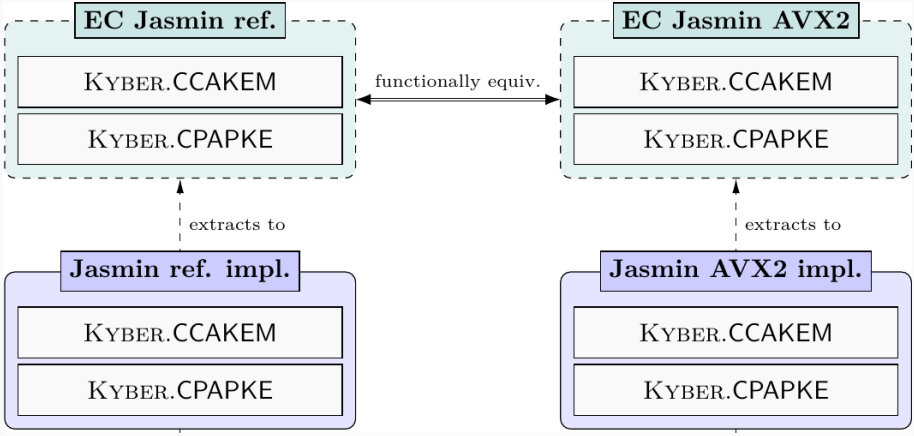


Functional correctness of ML-KEM implementations



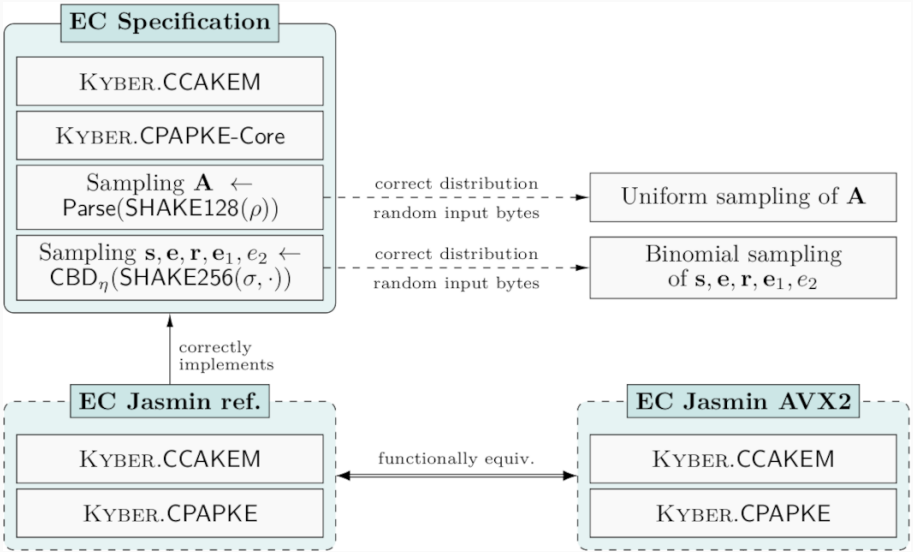
Almeida, Barbosa, Barthe, Grégoire, Laporte, Léchenet, Oliveira, Pacheco, Quaresma, Schwabe, Séré, and Strub. *Formally verifying Kyber – Episode IV: Implementation Correctness.* TCHES 2023-3.

Functional correctness of ML-KEM 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 ML-KEM implementations



From Kyber to ML-KEM

Almeida, Arranz Olmos, Barbosa, Barthe, Dupressoir, Grégoire, Laporte, Léchenet, Low, Oliveira, Pacheco, Quaresma, Schwabe, Strub. *Formally verifying Kyber – Episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt* CRYPTO 2024.

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

[3] 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[3]

[3] 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

Performance of verified ML-KEM-768 code

Implementation	operation	8700K	11700K	13900K
C/asm AVX2	keygen	39722	36958	31448
	encaps	39761	38082	32090
	decaps	46161	42566	36064
Jasmin AVX2	keygen	40134	37458	34732
	encaps	40599	37798	35212
	decaps	43437	39970	43784

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);
  ...
}
```

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 `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

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?

Selective SLH

Do we need to mask/protect all loads?

- No need to mask loads into registers that never enter leaking instructions

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

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
- 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 = #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`

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
- Overhead for ML-KEM-768 (on Intel 11700K):
  - 1.73% for Keypair
  - 1.60% for Encaps
  - 1.50% for Decaps

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

How about other Spectre variants?

Spectre v2

- Exploits speculation of indirect branches
- Jasmin does not support indirect branches

How about other Spectre variants?

Spectre v2

- Exploits speculation of indirect branches
- Jasmin does not support indirect branches

Spectre v3

- Better known as Meltdown
- Hardware bug, fixed in Hardware/Firmware

How about other Spectre variants?

Spectre v2

- Exploits speculation of indirect branches
- Jasmin does not support indirect branches

Spectre v3

- Better known as Meltdown
- Hardware bug, fixed in Hardware/Firmware

Spectre v4

- "Speculative store bypass"
- Loads may speculatively retrieve stale data
- Disable with SSBD CPU flag
- Overhead of around 5% for Kyber-768

But wait, there's more: Spectre-RSB

The attack

- Function returns use return-stack buffer (RSB) for speculative execution
- "Speculatively return to address on top of RSB"

But wait, there's more: Spectre-RSB

The attack

- Function returns use return-stack buffer (RSB) for speculative execution
- "Speculatively return to address on top of RSB"
- RSB is shared between processes running on the same core
- By default, RSB is not "wiped" on context switch

But wait, there's more: Spectre-RSB

The attack

- Function returns use return-stack buffer (RSB) for speculative execution
- "Speculatively return to address on top of RSB"
- RSB is shared between processes running on the same core
- By default, RSB is not "wiped" on context switch
- Attacker can make returns jump **anywhere** (speculatively)

But wait, there's more: Spectre-RSB

The attack

- Function returns use return-stack buffer (RSB) for speculative execution
- "Speculatively return to address on top of RSB"
- RSB is shared between processes running on the same core
- By default, RSB is not "wiped" on context switch
- Attacker can make returns jump **anywhere** (speculatively)

High-level countermeasure idea

- Limit attacker capabilities
- Speculative return only to well-defined restricted set of locations
- Use LFENCE or selective SLH to protect at those locations

Return tables and more security typing

- Jasmin compiler has global view
- For each function, compiler knows all call sites into this function

Arranz Olmos, Barthe, Chuengsatiansup, Grégoire, Laporte, Oliveira, Schwabe, Yarom, Zhang. *Protecting cryptographic code against Spectre-RSB* ePrint 2024/1070.

Return tables and more security typing

- Jasmin compiler has global view
- For each function, compiler knows all call sites into this function
- Replace return instructions with **return tables:**
  - Sequence of conditional branches to select return location
  - Number of branch instructions is logarithmic in number of call sites

Arranz Olmos, Barthe, Chuengsatiansup, Grégoire, Laporte, Oliveira, Schwabe, Yarom, Zhang. *Protecting cryptographic code against Spectre-RSB* ePrint 2024/1070.

Return tables and more security typing

- Jasmin compiler has global view
- For each function, compiler knows all call sites into this function
- Replace return instructions with **return tables:**
  - Sequence of conditional branches to select return location
  - Number of branch instructions is logarithmic in number of call sites
- Effect: we speculatively "return" only to some call site of the respective function
- Speculation is now "Spectre v1" style (conditional branch)

Arranz Olmos, Barthe, Chuengsatiansup, Grégoire, Laporte, Oliveira, Schwabe, Yarom, Zhang. *Protecting cryptographic code against Spectre-RSB* ePrint 2024/1070.

Return tables and more security typing

- Jasmin compiler has global view
- For each function, compiler knows all call sites into this function
- Replace return instructions with **return tables:**
  - Sequence of conditional branches to select return location
  - Number of branch instructions is logarithmic in number of call sites
- Effect: we speculatively "return" only to some call site of the respective function
- Speculation is now "Spectre v1" style (conditional branch)
- Except, not quite:
  - Speculation of conditionals and loops is *within control-flow graph*
  - Misspeculation of function "return" is outside control-flow graph

Arranz Olmos, Barthe, Chuengsatiansup, Grégoire, Laporte, Oliveira, Schwabe, Yarom, Zhang. *Protecting cryptographic code against Spectre-RSB* ePrint 2024/1070.

Return tables and more security typing

- Jasmin compiler has global view
- For each function, compiler knows all call sites into this function
- Replace return instructions with **return tables:**
  - Sequence of conditional branches to select return location
  - Number of branch instructions is logarithmic in number of call sites
- Effect: we speculatively "return" only to some call site of the respective function
- Speculation is now "Spectre v1" style (conditional branch)
- Except, not quite:
  - Speculation of conditionals and loops is *within control-flow graph*
  - Misspeculation of function "return" is outside control-flow graph
- Need modifications to security type system:
  - `public` registers become `transient` after function call
  - In some situations, we can preserve `public` type

Arranz Olmos, Barthe, Chuengsatiansup, Grégoire, Laporte, Oliveira, Schwabe, Yarom, Zhang. *Protecting cryptographic code against Spectre-RSB* ePrint 2024/1070.

Security – zeroization

"... 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.

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

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.

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.

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.

Upcoming

Crypto Agent

- Spectre protections need to be *global*
- Move crypto primitives to separate process
- Implement (almost) this whole program in Jasmin

Upcoming

More efficient correctness proofs

- Kyber/ML-KEM proof was massive effort
- Scalability issue when considering multiple architectures
- Solution: better automation in EasyCrypt

Upcoming

Better constant-time support

- Currently: avoid secret branches and memory access
- Need to avoid also variable-time arithmetic (e.g., DIV)
- Principled solution also for future microarchitectures

https://formosa-crypto.org