MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

High-assurance crypto in practice – Challenges and latest results

Peter Schwabe

September 11, 2023

MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

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)

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
- 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 . . . ?

PQC

- 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

Hertzbleed

CACHE OUT

Advanced microarchitectural side channels

https://www.metal-archives.com/bands/Downfall/3540377075

Advanced microarchitectural side channels

MELTDOWN

Hertzbleed

CACHE OUT

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[1]

- **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

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**

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

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

High-assurance crypto

Formosan black bear

文A 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

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

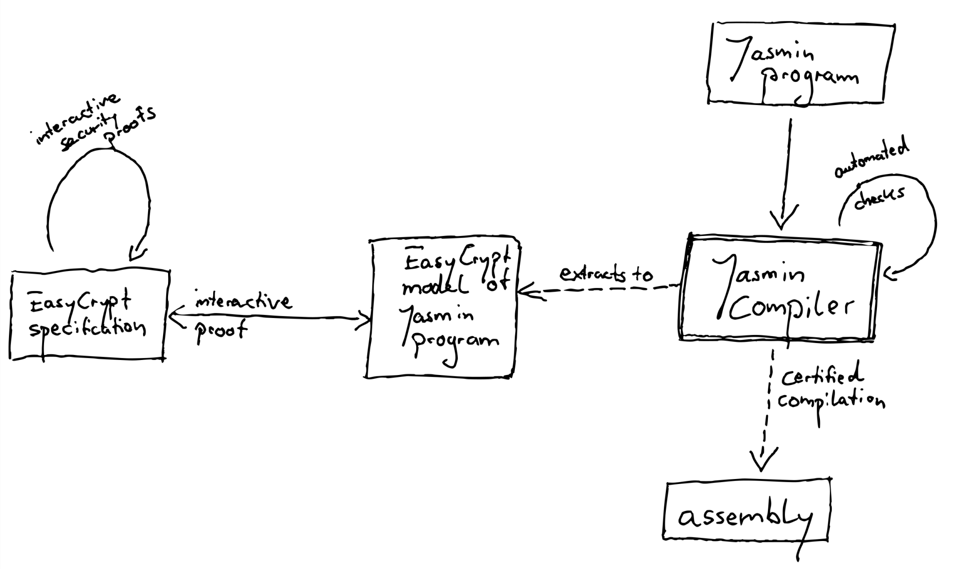
FORMOSA CRYPTO

- 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 >180 people

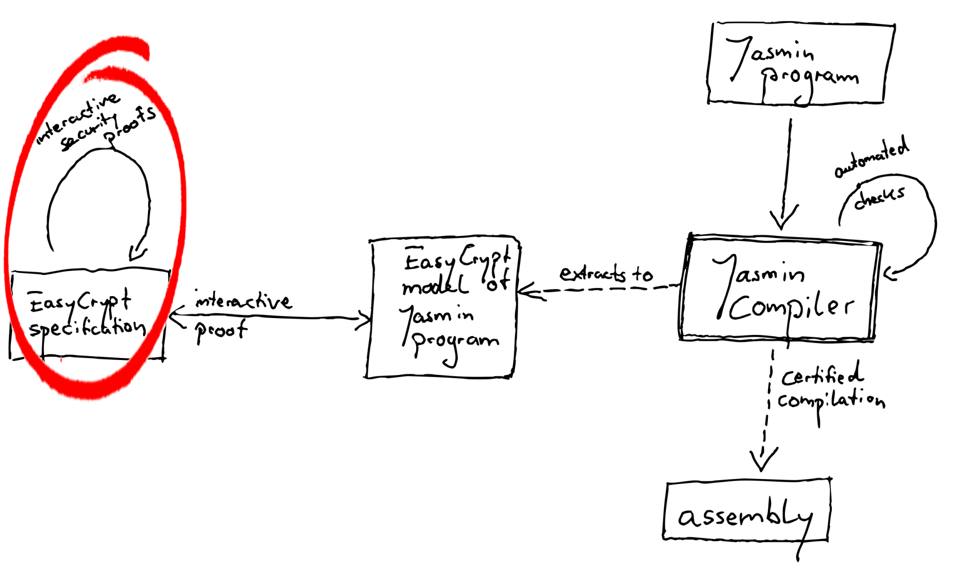
Joint work with. . .

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

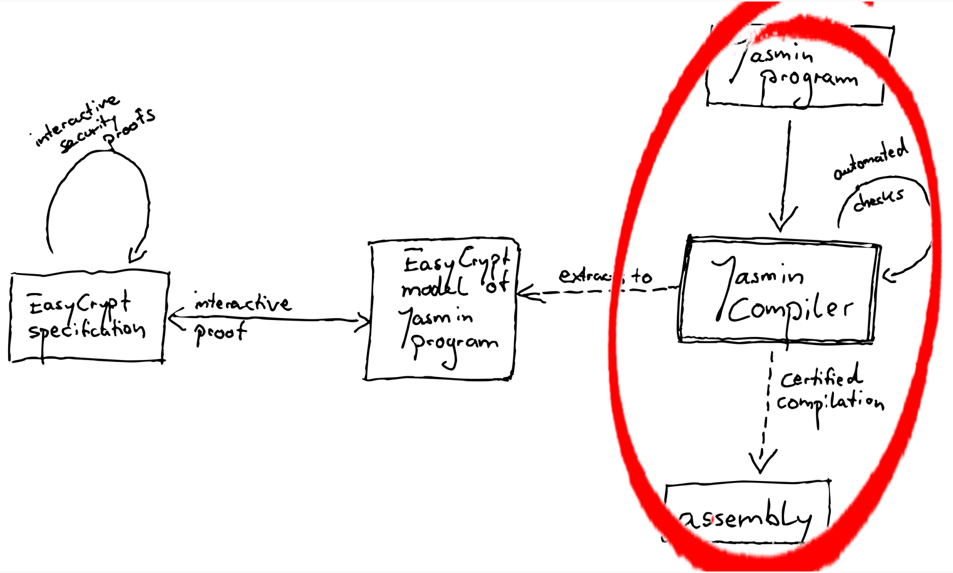
The toolchain and workflow



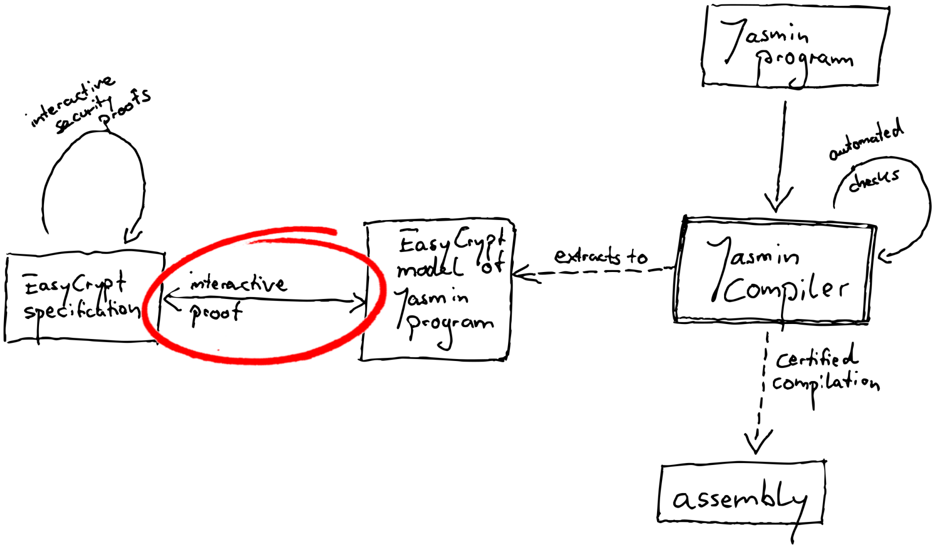
The toolchain and workflow



The toolchain and workflow



The toolchain and workflow



PQC security proofs in EasyCrypt

Barbosa, Barthe, Fan, Grégoire, Hung, Katz, Strub, Wu, and Zhou. *EasyPQC: Verifying Post-Quantum Cryptography.* ACM CCS 2021

PQC security proofs in EasyCrypt

Barbosa, Barthe, Fan, Grégoire, Hung, Katz, Strub, Wu, and Zhou. *EasyPQC: Verifying Post-Quantum Cryptography.* ACM CCS 2021

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

PQC security proofs in EasyCrypt

Barbosa, Barthe, Fan, Grégoire, Hung, Katz, Strub, Wu, and Zhou. *EasyPQC: Verifying Post-Quantum Cryptography.* ACM CCS 2021

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

Barbosa, Barthe, Fan, Grégoire, Hung, Katz, Strub, Wu, and Zhou. *EasyPQC: Verifying Post-Quantum Cryptography.* ACM CCS 2021

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

Barbosa, Dupressoir, Grégoire, Hülsing, Meijers, and Strub. *Machine-Checked Security for XMSS as in RFC 8391 and SPHINCS^{+}.* 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

Jasmin – assembly in your head

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 → 1 line in assembly
  - A few exceptions, but highly predictable
  - Compiler does not schedule code
  - Compiler does not spill registers

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

Jasmin – assembly in your head

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 → 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]

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

Jasmin – assembly in your head

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 → 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

Jasmin “Hello World!”

C code

```c
#include <stdio.h>

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

Jasmin code

Jasmin “Hello World!”

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!"

```
export fn add42(reg u64 x) -> reg u64 {
  reg u64 r;
  r = x;
  r += 42;
  return r;
}
```

Jasmin “Hello World!”

https://cryptojedi.org/programming/jasmin.shtml

Jasmin “Hello World!”

```
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;
}
```

Jasmin “Hello World!”

```
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;
}
```

Jasmin "Hello World!"

```
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

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);
  ...
}
```

It's more subtle than this

```
fn aes_rounds (stack u128[11] rkeys, reg u128 in) -> reg u128 {
  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!

Ammanaghatta Shivakumar, Barnes, Barthe, Cauligi, Chuengsatiansup, Genkin, O'Connell, Schwabe, Sim, and Yarom. *Spectre Declassified: Reading from the Right Place at the Wrong Time.* IEEE S&P 2023.

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

Do we need to mask/protect all loads?

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

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

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

Performance impact (Comet Lake cyles)

Primitive	Impl.	Op.	CT	SCT	overhead [%]
ChaCha20	avx2	32 B	314	352	12.10
	avx2	32 B xor	314	352	12.10
	avx2	128 B	330	370	12.12
	avx2	128 B xor	338	374	10.65
	avx2	1 KiB	1190	1234	3.70
	avx2	1 KiB xor	1198	1242	3.67
	avx2	1 KiB	18872	18912	0.21
	avx2	16 KiB xor	18970	18994	0.13

Performance impact (Comet Lake cyles)

Primitive	Impl.	Op.	CT	SCT	overhead [%]
Poly1305	avx2	32 B	46	78	69.57
	avx2	32 B verif	48	84	75.00
	avx2	128 B	136	172	26.47
	avx2	128 B verif	140	170	21.43
	avx2	1 KiB	656	686	4.57
	avx2	1 KiB verif	654	686	4.89
	avx2	16 KiB	8420	8450	0.36
	avx2	16 KiB verif	8416	8466	0.59

Performance impact (Comet Lake cyles)

Primitive	Impl.	Op.	CT	SCT	overhead [%]
X25519	mulx	smult	98352	98256	-0.098
	mulx	base	98354	98262	-0.094
Kyber512	avx2	keypair	25694	25912	0.848
	avx2	enc	35186	35464	0.790
	avx2	dec	27684	27976	1.055
Kyber768	avx2	keypair	42768	42888	0.281
	avx2	enc	54518	54818	0.550
	avx2	dec	43824	44152	0.748

Correctness – memory and thread safety, termination

- No global variables $\rightarrow$ thread safety

Correctness – memory and thread safety, termination

- No global variables $\rightarrow$ 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 $\rightarrow$ thread safety
- **Static** safety checker:
  - Uses language limitations
  - Ensures termination
  - Ensures memory safety (and prints conditions for inputs)
  - Not part of "standard compilation": `-checksafety`
- 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`)

Correctness – the missing link

Wednesday, 11:25, Track 1 (South Hall 2): Miguel Quaresma. *Formally verifying Kyber – Episode IV: Implementation Correctness.*

Correctness – the missing link

Wednesday, 11:25, Track 1 (South Hall 2): Miguel Quaresma. *Formally verifying Kyber – Episode IV: Implementation Correctness.*

"I'm carefully optimistic that we have the full proof and optimized software done by summer."

—me, May 2020

Correctness – the missing link

Wednesday, 11:25, Track 1 (South Hall 2): Miguel Quaresma. *Formally verifying Kyber – Episode IV: Implementation Correctness.*

"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

Correctness – the missing link

Wednesday, 11:25, Track 1 (South Hall 2): Miguel Quaresma. *Formally verifying Kyber – Episode IV: Implementation Correctness.*

"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...

[5] 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)[5]
  - (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!**

[5]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, needs users!
- 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, 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!

Resources

Join the effort:

https://formosa-crypto.org

Use the results:

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