Radboud University

Engineering Cryptographic Software

High-assurance crypto and Jasmin

Winter 2024/25

This course

1. Limit crypto to “secure-channel” crypto
2. Make crypto software fast and secure (against software attacks)

This course

1. Limit crypto to “secure-channel” crypto
2. Make crypto software fast and secure (against software attacks)

Secure-channel crypto

- Symmetric crypto (block/stream cipher, hash function, MAC)
- ECC (ECDH, Schnorr signatures)

This course

1. Limit crypto to "secure-channel" crypto
2. Make crypto software fast and secure (against software attacks)

Secure-channel crypto

- Symmetric crypto (block/stream cipher, hash function, MAC)
- ECC (ECDH, Schnorr signatures)

Fast and secure

- Optimize in C/assembly
- Follow the "constant-time" paradigm

Some things I like about the course

- Low-level programming on predictable platform
- Algorithmics of multiprecision arithmetic
- Scalar-multiplication algorithms

The times they are a-changin'

Post-quantum crypto

- The world is moving from ECC to PQC
- (Almost) no need for multiprecision arithmetic
- (Almost) no need for scalar multiplication
- More complex than optimizing just ECC scalar multiplication
- Starting to become part of standard crypto courses

The times they are a-changin'

Advanced microarchitectural attacks

- Spectre: exploit leaks in speculative execution
- Hertzbleed: power leakage translates to frequency changes
- Augury+GoFetch: attacks exploiting data-dependent prefetchers
- Three categories of CPUs/platforms:
  - Platforms that support secure implementations (e.g., OpenTitan)
  - Platforms that actively make secure implementations hard (large general-purpose CPUs)
  - Platforms that are in between (e.g., Cortex-M4)

The times they are a-changin'

High-assurance crypto

- Formal methods to improve crypto (software)
- Possibly move away from C/assembly

3 properties for crypto code

1. Correctness

- Functionally correct
- Memory safety
- Thread safety
- Termination

2. Security

- Don't leak secrets
- "Constant-time"
- Resist Spectre attacks
- Resist Power/EM attacks
- Fault protection
- Easy-to-use APIs

3. Efficiency

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

The “traditional approach”

1. Implement crypto in C
2. Identify most relevant parts for performance
3. Re-implement those in assembly

Correctness?

“Are you actually sure that your software is correct?”

—prof. Gerhard Woeginger, Jan. 24, 2011.

#epicfail

```
mulq  crypto_sign_ed25519_amd64_64_38
add  %rax,%r13
adc %rdx,%r14
adc $0,%r14
mov  %r9,%rax
mulq  crypto_sign_ed25519_amd64_64_38
add  %rax,%r14
adc %rdx,%r15
adc $0,%r15
mov  %r10,%rax
mulq  crypto_sign_ed25519_amd64_64_38
add  %rax,%r15
adc %rdx,%rbx
adc $0,%rbx
mov  %r11,%rax
mulq  crypto_sign_ed25519_amd64_64_38
add  %rax,%rbx
mov  $0,%rsi
adc %rdx,%rsi
```

- Code snippet is from > 8000 lines of assembly
- Crypto **always** has more possible inputs than we can exhaustively test
- Some bugs are triggered with very low probability
- Testing won't catch these bugs
- Audits might, but this requires expert knowledge!

Security?

Traditional timing attacks

- Software only, can be carried out remotely
- In principle, we know how to systematically avoid them
- Increasingly standard requirement: “constant-time”

Security?

Traditional timing attacks

- Software only, can be carried out remotely
- In principle, we know how to systematically avoid them
- Increasingly standard requirement: “constant-time”

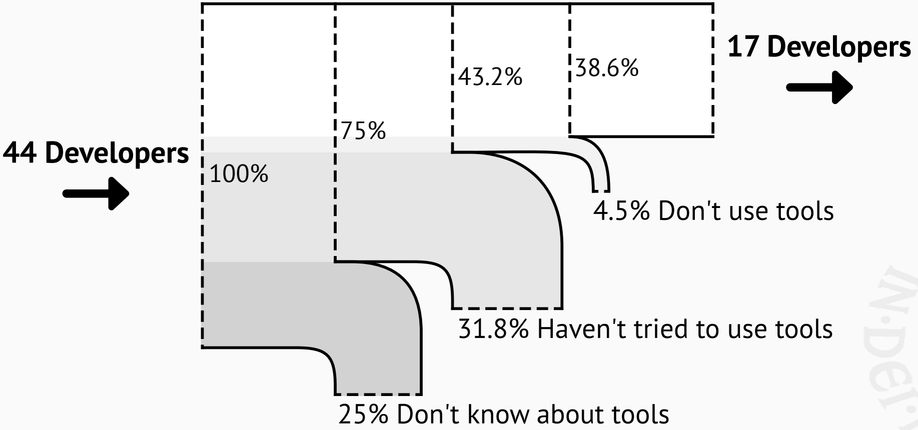
Plus side

- Full control (at least for assembly)
- Various tools to check for timing leaks

Minus side

- Many ways to screw up
- C compiler isn’t built for crypto

Security?



Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar: *"They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks*. IEEE S&P 2022

3. Efficiency!

High-assurance crypto

Traditional approach is great at producing very efficient software that is neither (guaranteed to be) correct nor (guaranteed to be) secure.

High-assurance crypto

Traditional approach is great at producing very efficient software that is neither (guaranteed to be) correct nor (guaranteed to be) secure.

- Idea: Use tools/techniques from formal methods to prove
  - functional correctness (including e.g., safety);
  - certain implementation security properties; (and
  - cryptographic security through reductions)

High-assurance crypto

Traditional approach is great at producing very efficient software that is neither (guaranteed to be) correct nor (guaranteed to be) secure.

- Idea: Use tools/techniques from formal methods to prove
  - functional correctness (including e.g., safety);
  - certain implementation security properties; (and
  - cryptographic security through reductions)
- Crypto software is a special here in multiple ways:
  - Usually fairly little code (**+**)
  - Has precise formal specification (**+**)
  - Inherently security-critical (**+**)

High-assurance crypto

Traditional approach is great at producing very efficient software that is neither (guaranteed to be) correct nor (guaranteed to be) secure.

- Idea: Use tools/techniques from formal methods to prove
  - functional correctness (including e.g., safety);
  - certain implementation security properties; (and
  - cryptographic security through reductions)
- Crypto software is a special here in multiple ways:
  - Usually fairly little code (**+**)
  - Has precise formal specification (**+**)
  - Inherently security-critical (**+**)
  - Highly performance critical (**–**)

High-assurance crypto

Traditional approach is great at producing very efficient software that is neither (guaranteed to be) correct nor (guaranteed to be) secure.

- Idea: Use tools/techniques from formal methods to prove
  - functional correctness (including e.g., safety);
  - certain implementation security properties; (and
  - cryptographic security through reductions)
- Crypto software is a special here in multiple ways:
  - Usually fairly little code (**+**)
  - Has precise formal specification (**+**)
  - Inherently security-critical (**+**)
  - Highly performance critical (**–**)

We want formal guarantees without giving up on performance.

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 >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

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]

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

Formosan black bear

Conservation status

Vulnerable (IUCN 3.1)[1]

https://en.wikipedia.org/wiki/Formosan_black_bear

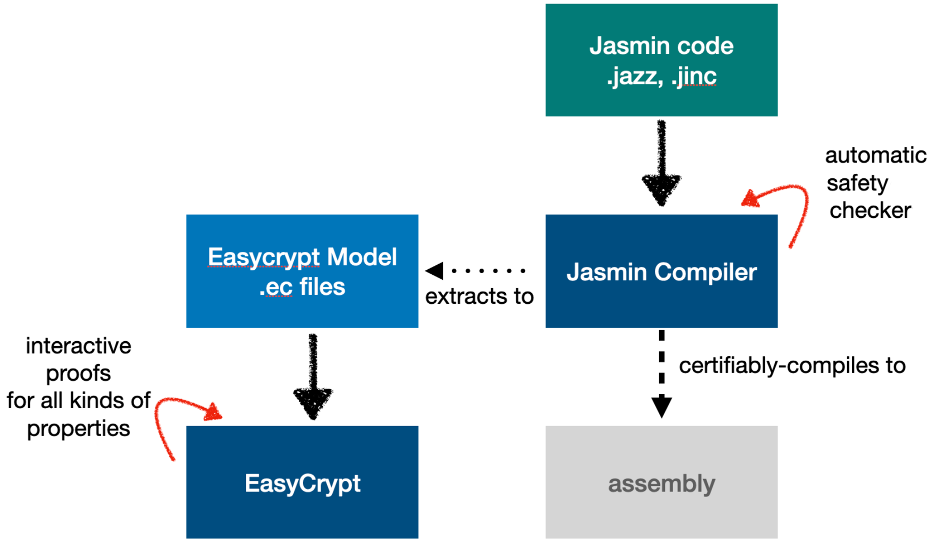
High-assurance crypto

FORMOSA CRYPTO

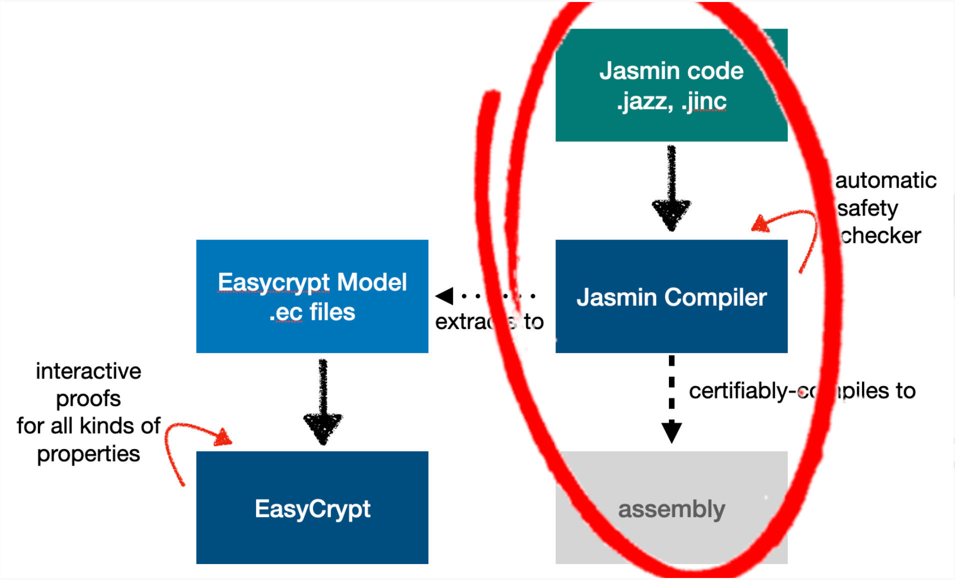
- Effort to formally verify crypto
- Goal: **verified PQC ready for deployment**
- Three main projects:
  - EasyCrypt proof assistant
  - Jasmin programming language
  - Libjade (PQ-)crypto library
- Core community of ≈ 30–40 people
- Discussion forum with >280 people

University of Bristol · CASA Cyber Security in the Age of Large-Scale Adversaries · IMDEA Software Institute · INESCTEC · Inria Inventeurs du monde numérique · Universität Klagenfurt · Max Planck Institute for Security and Privacy · U.Porto · Radboud University · Ruhr Universität Bochum · TU/e Eindhoven University of Technology

The toolchain and workflow



The toolchain and workflow



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**

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

Jasmin language features

- Functions with arguments and local variables
- Optionally: `inline` functions
- `export` functions to interface with C

Jasmin language features

- Functions with arguments and local variables
- Optionally: `inline` functions
- `export` functions to interface with C
- Stack and register arrays
- Loops (`while` and `for`)
- Conditionals (`if`, `else`)

Jasmin language features

- Functions with arguments and local variables
- Optionally: `inline` functions
- `export` functions to interface with C
- Stack and register arrays
- Loops (`while` and `for`)
- Conditionals (`if`, `else`)
- “Intrinsics”, e.g., for vector instructions
- Limited support for system calls

Memory and thread safety

- Jasmin does not support dynamic memory allocation
- All memory locations are either
  - external memory accessible through `export` function pointer arguments, or
  - allocated on the stack

Memory and thread safety

- Jasmin does not support dynamic memory allocation
- All memory locations are either
  - external memory accessible through **`export`** function pointer arguments, or
  - allocated on the stack
- Checking memory safety is separate compiler pass

```
jasminc -checksafety INPUT.jazz
```

- This typically takes a while to finish
- Safety checker is currently being re-written

Memory and thread safety

- Jasmin does not support dynamic memory allocation
- All memory locations are either
  - external memory accessible through `export` function pointer arguments, or
  - allocated on the stack
- Checking memory safety is separate compiler pass

```
jasminc -checksafety INPUT.jazz
```

- This typically takes a while to finish
- Safety checker is currently being re-written
- Jasmin does not have global variables
- Thread safe (except if external memory is shared)

So, where are we?

Correctness

- Functional correctness through EasyCrypt proofs
- Thread and **memory safety** guaranteed by Jasmin

Efficiency

Security

So, where are we?

Correctness

- Functional correctness through EasyCrypt proofs
- Thread and **memory safety** guaranteed by Jasmin
- Still need to check that EC specification is correct!
- Could be addressed by machine-readable standards

Efficiency

Security

So, where are we?

Correctness

- Functional correctness through EasyCrypt proofs
- Thread and **memory safety** guaranteed by Jasmin
- Still need to check that EC specification is correct!
- Could be addressed by machine-readable standards

Efficiency

- Some limitations compared to assembly for memory safety
- No limitations that (majorly) impact performance

Security

So, where are we?

Correctness

- Functional correctness through EasyCrypt proofs
- Thread and **memory safety** guaranteed by Jasmin
- Still need to check that EC specification is correct!
- Could be addressed by machine-readable standards

Efficiency

- Some limitations compared to assembly for memory safety
- No limitations that (majorly) impact performance

Security

- ???

Timing attacks

```
if(secret)
{
  do_A();
}
else
{
  do_B();
}
```

```
table[secret]
```

Constant-time: Avoid those!

Did we get it right?

Option 1: Auditing

"Originally, me, a glass of bourbon, and gdb were a good trio. But that got old pretty quick. (The manual analysis part – not the whiskey.)"

—Survey response in https://ia.cr./2021/1650

Did we get it right?

Option 2: Check/verify

- Implement, use tool to check "constant-time" property
- Tool overview by Ján Jančár: `https://crocs-muni.github.io/ct-tools/`
- Problems in practice:
  - Some tools not sound
  - Some tools not on binary/asm level
  - Some tools not usable

} Fairly high on my whishlist. . .

Did we get it right?

Option 3: Avoid variable-time code

- Prevent leaking patterns on source level
- Prove that compilation doesn't introduce leakage

Information-flow type system

- 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"

Information-flow type system

- 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`

Information-flow type system

- 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)

Information-flow type system

- 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)
- **Jasmin compiler is verified to preserve constant-time!**

Gilles Barthe, Benjamin Gregoire, Vincent Laporte, and Swarn Priya. *Structured Leakage and Applications to Cryptographic Constant-Time and Cost*. CCS 2021. https://eprint.iacr.org/2021/650

Information-flow type system

- 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)
- **Jasmin compiler is verified to preserve constant-time!**
- Explicit `#declassify` primitive to move from `secret` to `public`
- `#declassify` creates a proof obligation!

Gilles Barthe, Benjamin Gregoire, Vincent Laporte, and Swarn Priya. *Structured Leakage and Applications to Cryptographic Constant-Time and Cost*. CCS 2021. https://eprint.iacr.org/2021/650

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);
  ...
}
```

Countermeasures

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 noticable 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 of 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 of 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 of 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

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 of 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

Type system supports programmer in writing efficient Spectre-v1-protected code!

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

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

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.

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.

Summary

Programming in Jasmin gives you

- A more convenient way to “write assembly”
- Safety guarantees
- Systematic timing-attack protection
- Systematic Spectre v1 protection
- Link to computer-verified (EasyCrypt) proofs of
  - Functional correctness
  - Cryptographic security

Summary

Programming in Jasmin gives you

- A more convenient way to "write assembly"
- Safety guarantees
- Systematic timing-attack protection
- Systematic Spectre v1 protection
- Link to computer-verified (EasyCrypt) proofs of
  - Functional correctness
  - Cryptographic security
- Spoiler: there's more to come

Join us!

https://formosa-crypto.org

https://formosa-crypto.zulipchat.com/

Research internship opportunity

- Location: Rennes, France
- Topic: Systematic Analysis of Side Channels in Novel ARM Microarchitectures
- Researchers: Daniel De Almeida Braga & Thomas Rokicki
- Possibility to follow up with Ph.D. trajectory