



Engineering Cryptographic Software

High-assurance crypto and jasmin

Winter 2022



This course

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



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'

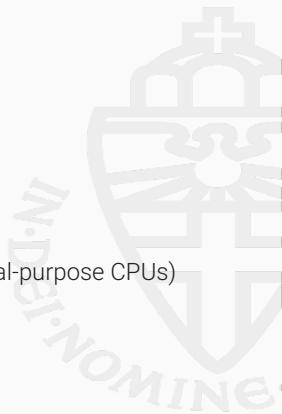
Post-quantum crypto

- 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
- Not (yet) part of standard crypto courses



Advanced microarchitectural attacks

- Attacks exploit leaks in speculative execution
- Hertzbleed: power leakage translates to frequency changes
- 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

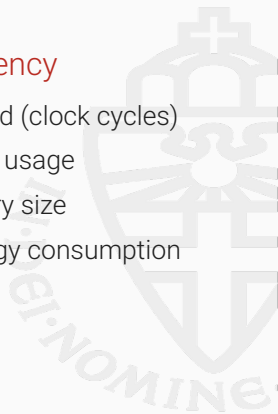
- Functionally correct
- Memory safety
- Thread safety
- Termination

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



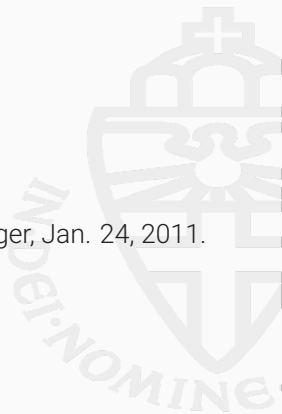
The “traditional approach”

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



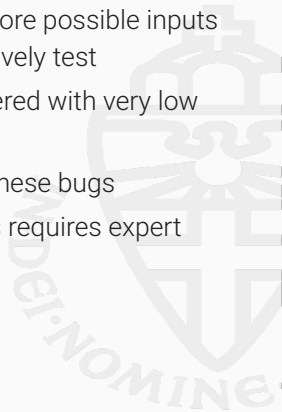
"Are you actually sure that your software is correct?"

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



```
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!



Timing attacks

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



Timing attacks

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

Plus side

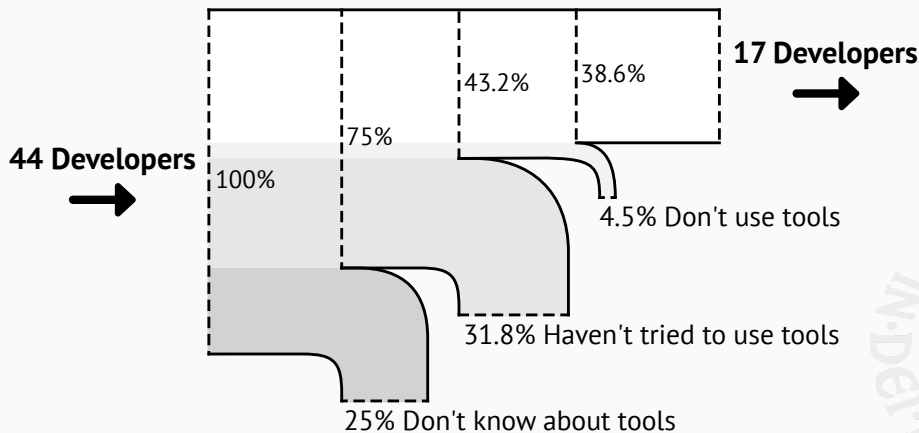
- Full control (at least for assembly)
- Various tools to check for timing leaks

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!



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



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)



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 special here in multiple ways:
 - Usually fairly little code (+)
 - Has precise formal specification (+)
 - Inherently security-critical (+)



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 (-)



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 (-)

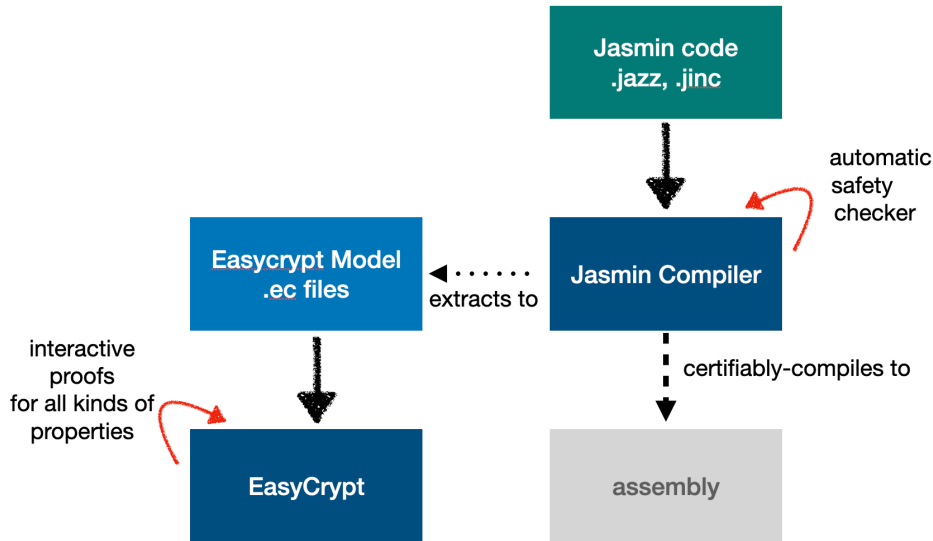
We want formal guarantees without giving up on performance.



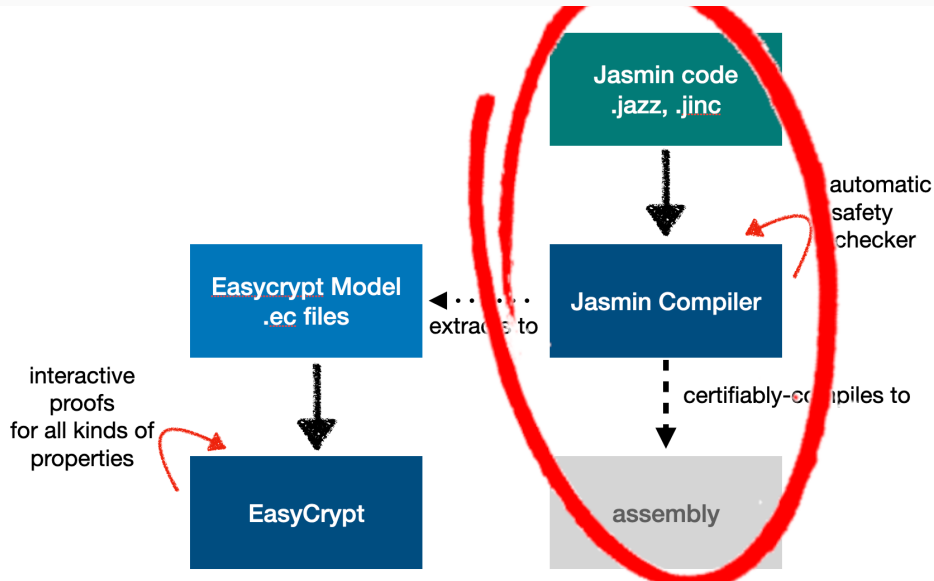
- Effort to formally verify crypto
- Currently three main projects:
 - EasyCrypt proof assistant
 - jasmin programming language
 - libjade (PQ-)crypto library
- Core community of ≈ 30 –40 people
- Discussion forum with >100 people



The toolchain and workflow



The toolchain and workflow



Jasmin – assembly in your head

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, Pierre-Yves 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 asm
 - A few exceptions, but highly predictable
 - Compiler does not schedule code
 - Compiler does not spill registers



Jasmin – assembly in your head

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, Pierre-Yves 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 asm
 - 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



Jasmin – assembly in your head

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, Pierre-Yves 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 asm
 - 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
- Many new features since 2017 paper!



C code

```
#include <stdio.h>

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

jasmin code



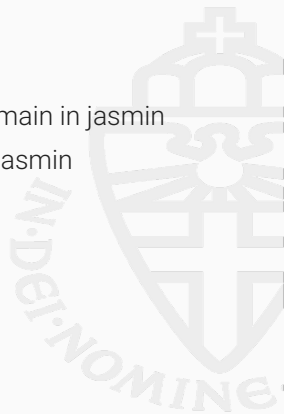
C code

```
#include <stdio.h>

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

jasmin code

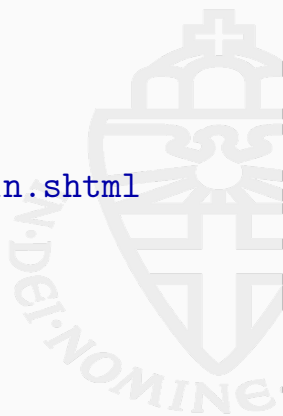
- We don't implement main in jasmin
- We don't have I/O in jasmin



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



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



Registers, stack, and arrays

- For each variable you need to decide if it is
 - living in a register: `reg`,
 - living on the stack: `stack`, or
 - replaced by immediates during compilation: `inline int`
- Integer types are called `u64`, `u32`, etc.
- Jasmin supports arrays of `reg` and `stack` variables:
 - `reg u32[10] a;`
 - `stack u64[100] b;`
- Arrays have **fixed** length
- Jasmin supports sub-arrays with fixed offsets and lengths, e.g. `b[16:32]` is the subarray of length 32 starting at index 16



- Conditionals (`if`, `else`) like in C



- Conditionals (`if`, `else`) like in C
- Two kinds of loops: `for` and `while`



- Conditionals (`if`, `else`) like in C
- Two kinds of loops: `for` and `while`
- `for` loops are automatically unrolled
- `for` iterate over an `inline int`



- Conditionals (`if`, `else`) like in C
- Two kinds of loops: `for` and `while`
- `for` loops are automatically unrolled
- `for` iterate over an `inline int`
- `while` loops are *real* loops with branch



Three kinds of “functions”

export functions

- Entry points into jasmin-generated code
- Need at least one `export` function in a jasmin program
- Follows (Linux) AMD64 C function-call ABI



Three kinds of “functions”

export functions

- Entry points into jasmin-generated code
- Need at least one `export` function in a jasmin program
- Follows (Linux) AMD64 C function-call ABI

inline functions

- Historically only non-`export` functions
- Can receive stack-array arguments



Three kinds of “functions”

`export` functions

- Entry points into jasmin-generated code
- Need at least one `export` function in a jasmin program
- Follows (Linux) AMD64 C function-call ABI

`inline` functions

- Historically only non-`export` functions
- Can receive stack-array arguments

“Regular” functions

- Array arguments passed through `reg ptr`
- `reg ptr` cannot be modified through arithmetic
- No fixed function-call ABI (compilation has global view)
- Stack pointer decreased **by caller**



Jasmin errors

- Easy case: syntax errors



Jasmin errors

- Easy case: syntax errors
- Slightly tougher: missing casts, see, e.g.,
`t0 = a.[u256 (int)(32 *64u i)];`



Jasmin errors

- Easy case: syntax errors
- Slightly tougher: missing casts, see, e.g.,
t0 = a.[u256 (int)(32 *64u i)];
- Most time-consuming to debug: register-allocation errors
- Example 1: constraints not satisfiable

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



- Easy case: syntax errors
- Slightly tougher: missing casts, see, e.g.,
`t0 = a.[u256 (int)(32 *64u i)];`
- Most time-consuming to debug: register-allocation errors
- Example 1: constraints not satisfiable

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

- Example 2: Running out of registers

```
"kem.jazz", line 14 (1) to line 27 (1):  
compilation error:
```

```
register allocation: variable shkp.3135 must be allocated to conflicting register RSI { RSI.83 }  
make: *** [.../.../.../.../Makefile.common:73: kem.s] Error 1
```

- Register allocation is global
 - Changes at one place may cause allocation to fail somewhere else
 - Error messages not super-helpful



- Jasmin supports 128-bit XMM and 256-bit YMM registers: `u128` and `u256`
- Operations through “intrinsic”, e.g.,

```
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;  
}
```



AMD64 only

- Full functionality only for AMD64 assembly
- ARMv7M (Cortex-M4) support in development branch
- Future directions: ARMv8, RISC-V, OpenTitan



Some current limitations

AMD64 only

- Full functionality only for AMD64 assembly
- ARMv7M (Cortex-M4) support in development branch
- Future directions: ARMv8, RISC-V, OpenTitan

No “slice” arguments

- Arrays have to have fixed length also in function arguments
- Separate function for each input length, e.g.

```
fn _ishake256_128_33(reg ptr u8[128] out, reg const ptr u8[33] in) -> stack u8[128]
```

- **Not** an issue for variable-length arguments to **export** functions



Some current limitations

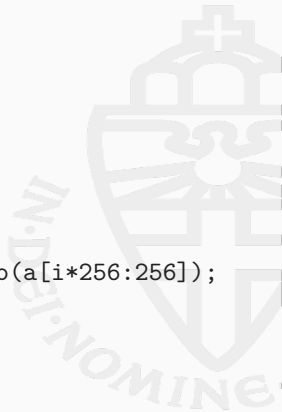
No register-indexed subarrays

This works

```
stack u16[768] a;
inline int i;
for i=0 to 3
{
    a[i*256:256] = foo(a[i*256:256]);
}
```

This does not

```
stack u16[768] a;
reg u64 i;
i = 0;
while(i < 3)
{
    a[i*256:256] = foo(a[i*256:256]);
    i += 1;
}
```

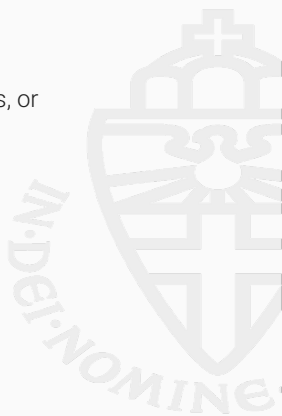


No typed export functions

- Inputs to `export` functions are of type `reg u64`
- Output is also a `reg u64`
- No argument passing over the stack
- No more than 6 arguments
- Distinguish between pointers and data only by usage/context



- 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

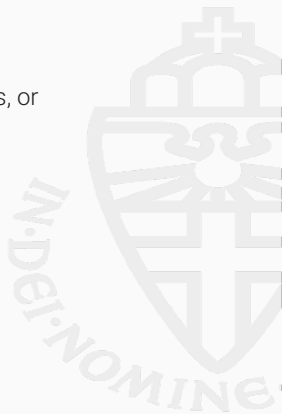


Memory and thread safety

- 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

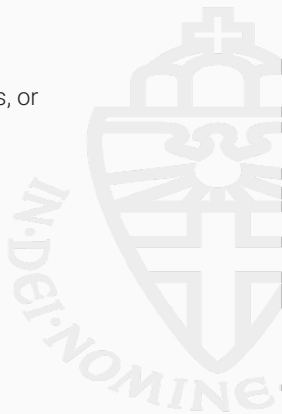
```
jasminc -checksafety INPUT.jazz
```

- This typically takes a while to finish



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
- 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 1: Auditing

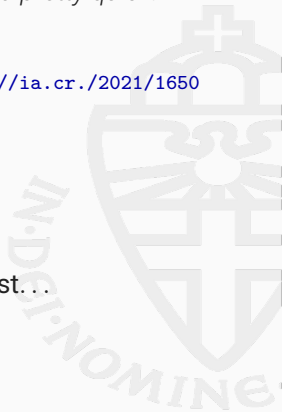
“Originally, me, a glass of bourbon, and gdb were a good trio. But that got old pretty quick. (The manual analysis part – not the whiskey.)”

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

Option 2: Check/verify

- Implement, use tool to check “constant-time” property
- Problems in practice:
 - Some tools not sound
 - Some tools not on binary/asm level
 - Some tools not usable

} Fairly high on my wishlist...



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>

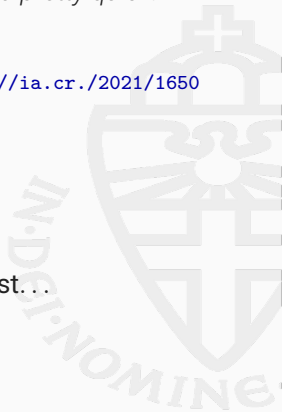
Option 2: Check/verify

- Implement, use tool to check “constant-time” property
- Problems in practice:
 - Some tools not sound
 - Some tools not on binary/asm level
 - Some tools not usable

} Fairly high on my wishlist...

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);
    ...
}
```



```
void victim_function(size_t x,  
                    size_t array1_size,  
                    const uint8_t *array1,  
                    const uint8_t *array2,  
                    uint8_t *temp)  
{  
    if (x < array1_size) {  
        *temp &= array2[array1[x] * 512];  
    }  
}
```



```
export fn victim_function(reg u64 x, reg u64 array1_size,  
                          reg u64 array1, reg u64 array2, reg u64 temp) {  
    reg u64 a;  
    reg u8 ab bb pab pbb t;  
    inline bool b;  
  
    t = (u8)[temp];  
    b = x < array1_size;  
    if (b) {  
        ab = (u8)[array1 + x];  
        a = (64u)ab;  
        a <<= 9;  
        bb = (u8)[array2 + a];  
        t &= bb;  
    }  
    (u8)[temp] = 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;
}
```



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!

Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O’Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom: *Spectre Declassified: Reading from the Right Place at the Wrong Time*. IEEE S&P 2023. <https://eprint.iacr.org/2022/426>



Fencing

- Can prevent speculation through **barriers** (LFENCE)
- Protecting *all* branches is possible but costly



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



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 noticeable 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 **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 **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 **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

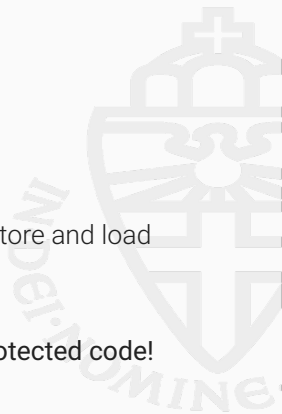
- We know what inputs **secret** and what inputs are **public**
- Most of the state is actually **secret**
- Most loads do not need `protect`!
- Even better: mark additional inputs as **secret**
- No cost of those inputs don't flow into leaking instructions
- Even better: Spills don't need `protect` 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 **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!



Performance results (Comet Lake cycles)

| 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 results (Comet Lake cycles)

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



Performance results (Comet Lake cycles)

| 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 |



- Spectre v1 is not the only speculative attack vector

Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, Lucas Tabary-Maujean: *Typing High-Speed Cryptography against Spectre v1*. <https://eprint.iacr.org/2022/1270>



Limitations

- Spectre v1 is not the only speculative attack vector
- Spectre v2: Avoid by not using indirect branches

Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, Lucas Tabary-Maujean: *Typing High-Speed Cryptography against Spectre v1*. <https://eprint.iacr.org/2022/1270>



Limitations

- Spectre v1 is not the only speculative attack vector
- Spectre v2: Avoid by not using indirect branches
- Spectre v4: Use SSBD: <https://github.com/tyhicks/ssbd-tools>

Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, Lucas Tabary-Maujean: *Typing High-Speed Cryptography against Spectre v1*. <https://eprint.iacr.org/2022/1270>



- Spectre v1 is not the only speculative attack vector
- Spectre v2: Avoid by not using indirect branches
- Spectre v4: Use SSBD: <https://github.com/tyhicks/ssbd-tools>
- **Our protection requires separation of crypto code!**
 - Typically crypto is living in the same address space as application
 - Any Spectre v1 gadget in application can leak keys!

Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, Lucas Tabary-Maujean: *Typing High-Speed Cryptography against Spectre v1*. <https://eprint.iacr.org/2022/1270>



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



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/>

