Engineering high-assurance crypto software

Peter Schwabe

September 30, 2022
Max-Planck Institute for Security and Privacy

• Founded in 2019
• Currently: 2 directors +
  • 2 directors
  • 6 (soon 8) research group leaders
  • ≈35 postdocs and Ph.D. students
• Long-term plan
  • 6 directors
  • 12 research group leaders
  • 200+ scientific staff
What crypto software (libraries) do you know?
What properties do you expect from crypto software?
3 properties

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
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?”

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?

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
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.
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.
Formosa Crypto

- 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

Easycrypt Model .ec files

EasyCrypt

Jasmin code .jazz, .jinc

Jasmin Compiler

assembly

interactive proofs for all kinds of properties

extracts to

certifiably-compiles to

automatic safety checker
The toolchain and workflow

- Jasmin code (.jazz, .jinc)
- Automatic safety checker
- Certifiably-compiles to assembly
- Interactively proofs for all kinds of properties
- EasyCrypt
- Easycrypt Model (.ec files)
- Extracts to
- Jasmin Compiler
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

• 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
• 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!

Jasmin – first steps

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
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
• 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
Loops and conditionals

• Conditionals (if, else) like in C
Loops and conditionals

• Conditionals (if, else) like in C
• Two kinds of loops: for and while
Loops and conditionals

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

• 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.,
  \[ t0 = a.[u256 (int)(32 \times 64u i)]; \]
Jasmin errors

- Easy case: syntax errors
- Slightly tougher: missing casts, see, e.g.,
  \[ t0 = a.\texttt{[u256 \texttt{(int)}(32 \times 64u i)]}; \]
- Most time-consuming to debug: register-allocation errors
- Example 1: constraints not satisfiable
  
  ```javascript
  export fn add42(reg u64 x) -> reg u64 {
    x += 42;
    return x;
  }
  ```

- Example 2: Running out of registers
  kemi.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 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;
  }
  ```
- 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
Vector computations

Scalar computation

• Load 32-bit integer $a$
• Load 32-bit integer $b$
• Perform addition $c \leftarrow a + b$
• Store 32-bit integer $c$

Vectorized computation

• Load 4 consecutive 32-bit integers $(a_0, a_1, a_2, a_3)$
• Load 4 consecutive 32-bit integers $(b_0, b_1, b_2, b_3)$
• Perform addition $(c_0, c_1, c_2, c_3) \leftarrow (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
• Store 128-bit vector $(c_0, c_1, c_2, c_3)$
Vector computations

Scalar computation

• Load 32-bit integer \(a\)
• Load 32-bit integer \(b\)
• Perform addition \(c \leftarrow a + b\)
• Store 32-bit integer \(c\)

Vectorized computation

• Load 4 consecutive 32-bit integers \((a_0, a_1, a_2, a_3)\)
• Load 4 consecutive 32-bit integers \((b_0, b_1, b_2, b_3)\)
• Perform addition
  \((c_0, c_1, c_2, c_3) \leftarrow (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)\)
• Store 128-bit vector \((c_0, c_1, c_2, c_3)\)

• Perform the same operations on independent data streams (SIMD)
• Vector instructions available on most “large” processors
• Instructions for vectors of bytes, integers, floats...
Vector computations

Scalar computation

- Load 32-bit integer \( a \)
- Load 32-bit integer \( b \)
- Perform addition \( c \leftarrow a + b \)
- Store 32-bit integer \( c \)

Vectorized computation

- Load 4 consecutive 32-bit integers \((a_0, a_1, a_2, a_3)\)
- Load 4 consecutive 32-bit integers \((b_0, b_1, b_2, b_3)\)
- Perform addition
  \[
  (c_0, c_1, c_2, c_3) \leftarrow (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)
  \]
- Store 128-bit vector \((c_0, c_1, c_2, c_3)\)

- Perform the same operations on independent data streams (SIMD)
- Vector instructions available on most “large” processors
- Instructions for vectors of bytes, integers, floats . . .
- Need to interleave data items (e.g., 32-bit integers) in memory
How fast is that?

• Consider the Intel Skylake processor with AVX2
How fast is that?

- Consider the Intel Skylake processor with AVX2
  - 32-bit load throughput: 2 per cycle
  - 32-bit add throughput: 4 per cycle
  - 32-bit store throughput: 1 per cycle

AVX2 vector instructions are almost as fast as scalar instructions but do $8 \times$ the work.

Situation on other architectures/microarchitectures is similar

Reason: cheap way to increase arithmetic throughput (less decoding, address computation, etc.)
How fast is that?

• Consider the Intel Skylake processor with AVX2
  • 32-bit load throughput: 2 per cycle
  • 32-bit add throughput: 4 per cycle
  • 32-bit store throughput: 1 per cycle
  • 256-bit load throughput: 2 per cycle
  • $8 \times$ 32-bit add throughput: 3 per cycle
  • 256-bit store throughput: 1 per cycle
How fast is that?

• Consider the Intel Skylake processor with AVX2
  • 32-bit load throughput: 2 per cycle
  • 32-bit add throughput: 4 per cycle
  • 32-bit store throughput: 1 per cycle
  • 256-bit load throughput: 2 per cycle
  • $8 \times 32$-bit add throughput: 3 per cycle
  • 256-bit store throughput: 1 per cycle

• AVX2 vector instructions are almost as fast as scalar instructions but do $8 \times$ the work
How fast is that?

- Consider the Intel Skylake processor with AVX2
  - 32-bit load throughput: 2 per cycle
  - 32-bit add throughput: 4 per cycle
  - 32-bit store throughput: 1 per cycle
  - 256-bit load throughput: 2 per cycle
  - $8 \times 32$-bit add throughput: 3 per cycle
  - 256-bit store throughput: 1 per cycle

- AVX2 vector instructions are almost as fast as scalar instructions but do $8 \times$ the work
- Situation on other architectures/microarchitectures is similar
- Reason: cheap way to increase arithmetic throughput (less decoding, address computation, etc.)
Vectorization in jasmin

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

```plaintext
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;
}
```
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.
  
  \[
  \text{fn } _\text{ishake256}_\text{128}_\text{33}(\text{reg ptr u8}\left[128\right] \text{out}, \text{reg const ptr u8}\left[33\right] \text{in}) \rightarrow \text{stack u8}\left[128\right]
  \]

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

  \[
  \text{fn } _\text{ishake256}_\text{128}_\text{33}(\text{reg ptr u8}[128] \text{ out}, \text{reg const ptr u8}[33] \text{ in}) -> \text{stack u8}[128]
  \]

- **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;
}
```
Some current limitations

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
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).
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
  
  ```
jasmnc -checksafety INPUT.jazz
  ```
- This typically takes a while to finish
• 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
• ???
if(secret)
{
    do_A();
}
else
{
    do_B();
}
• So, what do we do with code like this?

```plaintext
if s then
    r ← A
else
    r ← B
end if
```
Eliminating branches

• So, what do we do with code like this?
  
  \[
  \begin{align*}
  \text{if } s \text{ then} & \quad r \leftarrow A \\
  \text{else} & \quad r \leftarrow B \\
  \text{end if}
  \end{align*}
  \]

• Replace by

\[
  r \leftarrow sA + (1 - s)B
\]
• So, what do we do with code like this?
  
  if $s$ then
  
  $r \leftarrow A$
  
  else
  
  $r \leftarrow B$
  
  end if

• Replace by

  $$r \leftarrow sA + (1 - s)B$$

• Can expand $s$ to all-one/all-zero mask and use XOR instead of addition, AND instead of multiplication
Eliminating branches

• So, what do we do with code like this?

\[
\text{if } s \text{ then} \\
\quad r \leftarrow A \\
\text{else} \\
\quad r \leftarrow B \\
\text{end if}
\]

• Replace by

\[
r \leftarrow sA + (1 - s)B
\]

• Can expand \( s \) to all-one/all-zero mask and use XOR instead of addition, AND instead of multiplication

• For very fast \( A \) and \( B \) this can even be faster
Timing attacks – secret indices

table[secret]
uint32 table[TABLE_LENGTH];

uint32 lookup(size_t pos)
{
    size_t i;
    int b;
    uint32 r = table[0];
    for(i=1; i<TABLE_LENGTH; i++)
    {
        b = isequal(i, pos);
        cmov(&r, &table[i], b);
    }
    return r;
}
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
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

\[ \text{Fairly high on my wishlist...} \]

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!**

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!

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;
}
    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!

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 $\texttt{ms}$ (in a register)
- At every branch use arithmetic to update predicate
- Option 1: Mask every loaded value with $\texttt{ms}$
- Option 2: Mask every address with $\texttt{ms}$
- Effect: during misspeculation “leak” constant value
- Implemented in LLVM since version 8
  - Still noticable 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 $ms$:
  - $ms = \text{init}_\text{msf}()$: Translate to LFENCE, set $ms$ to zero
  - $ms = \text{set}_\text{msf}(b, ms)$: Set $ms$ according to branch condition $b$
    - Branches invalidate $ms$

- Two operations to lower level:
  - $x = \text{protect}(x, ms)$: Go from transient to public
    - $\text{protect}$ translates to mask by $ms$
  - $\text{declassify } r$: Go from secret to transient
    - $\text{declassify}$ 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 \( ms \):
  - \( ms = \#\text{init}_\text{msf}() \): Translate to LFENCE, set \( ms \) to zero
  - \( ms = \#\text{set}_\text{msf}(b, ms) \): Set \( ms \) according to branch condition \( b \)
  - Branches invalidate \( ms \)

- Two operations to lower level:
  - \( x = \#\text{protect}(x, ms) \): 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 cyles)

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Impl.</th>
<th>Op.</th>
<th>CT</th>
<th>SCT</th>
<th>overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChaCha20</td>
<td>avx2</td>
<td>32 B</td>
<td>314</td>
<td>352</td>
<td>12.10</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>32 B xor</td>
<td>314</td>
<td>352</td>
<td>12.10</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>128 B</td>
<td>330</td>
<td>370</td>
<td>12.12</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>128 B xor</td>
<td>338</td>
<td>374</td>
<td>10.65</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>1 KiB</td>
<td>1190</td>
<td>1234</td>
<td>3.70</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>1 KiB xor</td>
<td>1198</td>
<td>1242</td>
<td>3.67</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>1 KiB</td>
<td>18872</td>
<td>18912</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>16 KiB xor</td>
<td>18970</td>
<td>18994</td>
<td>0.13</td>
</tr>
</tbody>
</table>
Performance results (Comet Lake cyles)

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Impl.</th>
<th>Op.</th>
<th>CT</th>
<th>SCT</th>
<th>overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avx2</td>
<td>32 B</td>
<td>46</td>
<td>78</td>
<td>69.57</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>32 B verif</td>
<td>48</td>
<td>84</td>
<td>75.00</td>
</tr>
<tr>
<td>Poly1305</td>
<td>avx2</td>
<td>128 B</td>
<td>136</td>
<td>172</td>
<td>26.47</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>128 B verif</td>
<td>140</td>
<td>170</td>
<td>21.43</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>1 KiB</td>
<td>656</td>
<td>686</td>
<td>4.57</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>1 KiB verif</td>
<td>654</td>
<td>686</td>
<td>4.89</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>16 KiB</td>
<td>8420</td>
<td>8450</td>
<td>0.36</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>16 KiB verif</td>
<td>8416</td>
<td>8466</td>
<td>0.59</td>
</tr>
<tr>
<td>Primitive</td>
<td>Impl.</td>
<td>Op.</td>
<td>CT</td>
<td>SCT</td>
<td>overhead [%]</td>
</tr>
<tr>
<td>------------</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>---------</td>
<td>--------------</td>
</tr>
<tr>
<td>X25519</td>
<td>mulx</td>
<td>smult</td>
<td>98352</td>
<td>98256</td>
<td>-0.098</td>
</tr>
<tr>
<td></td>
<td>mulx</td>
<td>base</td>
<td>98354</td>
<td>98262</td>
<td>-0.094</td>
</tr>
<tr>
<td>Kyber512</td>
<td>avx2</td>
<td>keypair</td>
<td>25694</td>
<td>25912</td>
<td>0.848</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>enc</td>
<td>35186</td>
<td>35464</td>
<td>0.790</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>dec</td>
<td>27684</td>
<td>27976</td>
<td>1.055</td>
</tr>
<tr>
<td>Kyber768</td>
<td>avx2</td>
<td>keypair</td>
<td>42768</td>
<td>42888</td>
<td>0.281</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>enc</td>
<td>54518</td>
<td>54818</td>
<td>0.550</td>
</tr>
<tr>
<td></td>
<td>avx2</td>
<td>dec</td>
<td>43824</td>
<td>44152</td>
<td>0.748</td>
</tr>
</tbody>
</table>
Limitations

- Spectre v1 is not the only speculative attack vector

Limitations

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

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

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
- **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!

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/