Formosa Crypto

Peter Schwabe

December 5, 2022
“The public-key encryption and key-establishment algorithm that will be standardized is CRYSTALS-KYBER. The digital signatures that will be standardized are CRYSTALS-Dilithium, FALCON, and SPHINCS+. While there are multiple signature algorithms selected, NIST recommends CRYSTALS-Dilithium as the primary algorithm to be implemented”

—NIST IR 8413-upd1
Where are we?

Next steps for deployment

1. Take C/asm reference or optimized implementations.
2. Integrate into systems and protocols.
3. PQC migration done.
Next steps for deployment

1. Take C/asm reference or optimized implementations.
2. Integrate into systems and protocols.
3. PQC migration done.

Easier said than done – but is it even what we should be doing?
What else happened since 2016?
High-assurance crypto

- Computer-verified proofs of
  - functional correctness
  - implementation security
  - cryptographic security
- HACS workshop since 2016 (co-located with RWC)
What else happened since 2016?

High-assurance crypto

• Computer-verified proofs of
  • functional correctness
  • implementation security
  • cryptographic security

• HACS workshop since 2016 (co-located with RWC)

• High-assurance crypto software is used:
  • In the Linux kernel
  • In Firefox (NSS library)
  • In Google Chrome (BoringSSL)
High-assurance PQC

Formosa Crypto

- Formally verified open-source amazing crypto
- Effort to formally verify crypto
- Currently three main projects:
  - EasyCrypt proof assistant
  - jasmin programming language
  - libJade (PQ-)crypto library
- Core community of \( \approx \) 30–40 people
- Discussion forum with >100 people
The toolchain and workflow

EasyCrypt Model .ec files

Jasmin code .jazz, .jinc

Jasmin Compiler

eature safety checker

certifiably-compiles to assembly

interactive proofs for all kinds of properties

extracts to
libJade – Goals

• High-performance implementations of all NIST PQC primitives (first focus on Kyber and Dilithium)
• Multi-architecture support (first focus on AMD64)
• Easy “drop in” integration for most protocol libraries and systems
libJade – Goals

- High-performance implementations of all NIST PQC primitives (first focus on Kyber and Dilithium)
- Multi-architecture support (first focus on AMD64)
- Easy “drop in” integration for most protocol libraries and systems
- Automated proofs of thread safety and memory safety
- Certified compilation to assembly
libJade – Goals

• High-performance implementations of all NIST PQC primitives (first focus on Kyber and Dilithium)
• Multi-architecture support (first focus on AMD64)
• Easy “drop in” integration for most protocol libraries and systems
• Automated proofs of thread safety and memory safety
• Certified compilation to assembly
• Computer-verified (manual) proofs of functional correctness
• Connection to computer-verified (manual) cryptographic proofs
• High-performance implementations of all NIST PQC primitives (first focus on Kyber and Dilithium)
• Multi-architecture support (first focus on AMD64)
• Easy “drop in” integration for most protocol libraries and systems
• Automated proofs of thread safety and memory safety
• Certified compilation to assembly
• Computer-verified (manual) proofs of functional correctness
• Connection to computer-verified (manual) cryptographic proofs
• Verified resistance against well-defined classes of microarchitectural attacks
Mostly two rules

1. Don’t branch on secrets
2. Don’t access memory at secret locations
Traditional timing attacks

Mostly two rules

1. Don’t branch on secrets
2. Don’t access memory at secret locations

How do we check this?

1. Auditing – expensive, no guarantees
2. Constant-time checking tools; see https://crocs-muni.github.io/ct-tools/
3. Sound, easy to install/use, binary level
Mostly two rules

1. Don’t branch on secrets
2. Don’t access memory at secret locations

How do we check this?

1. Auditing – expensive, no guarantees
Traditional timing attacks

Mostly two rules

1. Don’t branch on secrets
2. Don’t access memory at secret locations

How do we check this?

1. Auditing – expensive, no guarantees
2. Constant-time checking tools; see https://crocs-muni.github.io/ct-tools/
   Sound, easy to install/use, binary level...?
Traditional timing attacks

Mostly two rules

1. Don’t branch on secrets
2. Don’t access memory at secret locations

How do we check this?

1. Auditing – expensive, no guarantees
2. Constant-time checking tools; see https://crocs-muni.github.io/ct-tools/
   Sound, easy to install/use, binary level...?
3. Constant-time source + secure compilation
Solution in jasmin/libJade (WIP)

- 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"
Solution in jasmin/libJade (WIP)

- 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
• 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)
Solution in jasmin/libJade (WIP)

- 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!

Solution in jasmin/libJade (WIP)

- 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!

Spectre v1 (“Speculative bounds-check bypass”)

```c
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);
    ...
}
```
    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 $ms$ (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
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.
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:
  - \texttt{secret}: secret
  - \texttt{public}: public, also during misspeculation
  - \texttt{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 if 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 if those inputs don’t flow into leaking instructions
- Even better: Spills don’t need **protect** if there is no branch between store and load
The special case of crypto

• We know what inputs **secret** and what inputs are **public**
• Most of the state is actually **secret**
• Most loads do not need **protect**!
• Even better: mark additional inputs as **secret**
• No cost if those inputs don’t flow into leaking instructions
• Even better: Spills don’t need **protect** if there is no branch between store and load
• Even better: “Spill” public data to MMX registers instead of stack
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 if those inputs don’t flow into leaking instructions
- Even better: Spills don’t need `protect` if there is no branch between store and load
- Even better: “Spill” public data to MMX registers instead of stack

Type system supports programmer in writing efficient Spectre-v1-protected code!
## Performance results (Comet Lake cycles)

<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 cycles)

<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>Poly1305</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></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>
</tbody>
</table>
## Performance results (Comet Lake cycles)

<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>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
- **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!

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