MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

High-assurance crypto – Part III: Jasmin

Peter Schwabe

January 30, 2023

So, where are we?

Efficiency/Speed ✓

Security ?

Correctness ?

Security?

We know what to do

- No secret-dependent branches
- No secret-dependent memory indexing
- No variable-time arithmetic on secrets

. . . **but how do we make sure we get it right?**

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!

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.

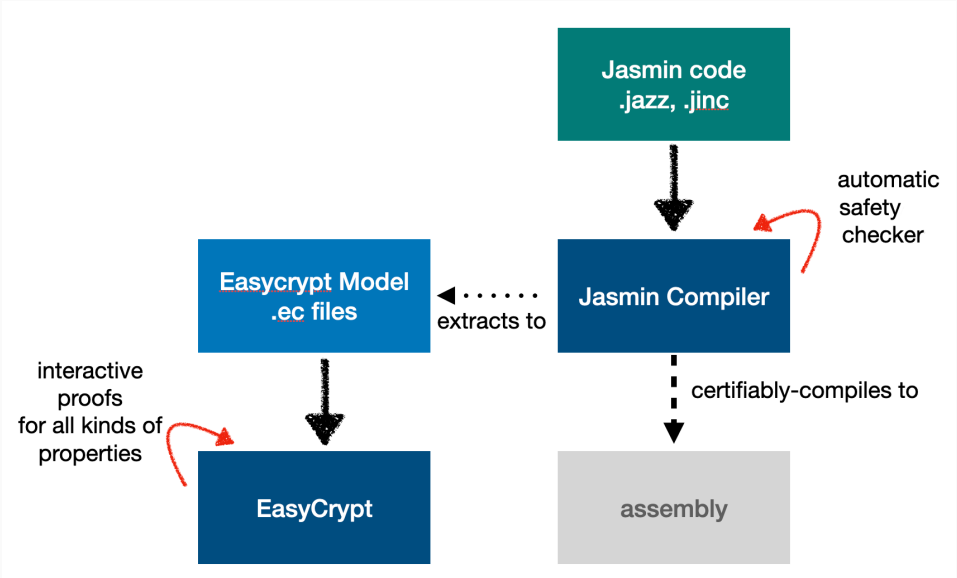
- Idea: Use tools/techniques from formal methods to prove
  - functional correctness (including e.g., safety);
  - certain implementation security properties; (and
  - cryptographic security through reductions)
- Crypto software is a special here in multiple ways:
  - Usually fairly little code (+)
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

We want formal guarantees without giving up on performance.

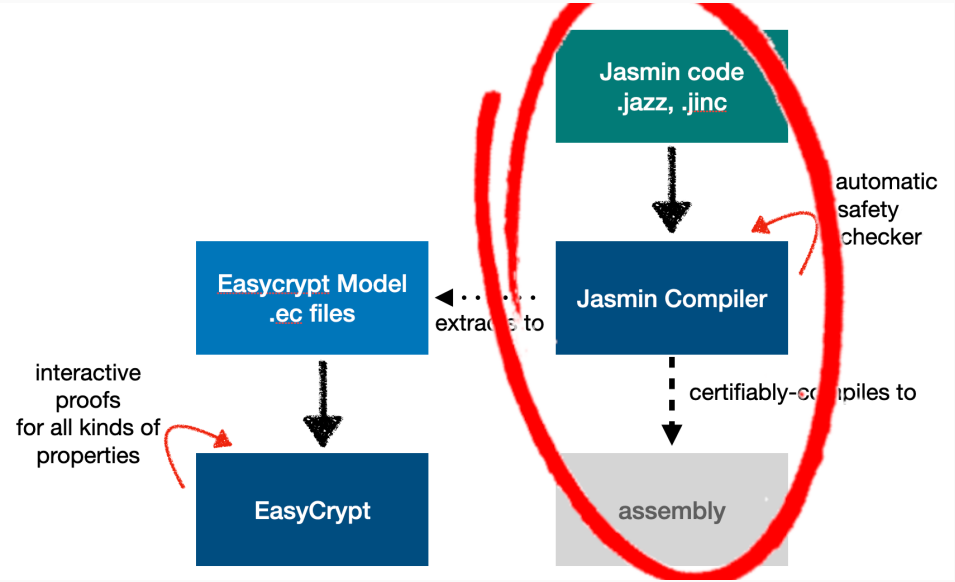
Formosa Crypto

- Effort to formally verify crypto
- Currently three main projects:
  - EasyCrypt proof assistant
  - jasmin programming language
  - libjade (PQ-)crpyto library
- Core community of $\approx$ 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 $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ 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

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

Jasmin – first steps

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

Jasmin – first steps

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

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 modifed 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
- Slighly tougher: missing casts, see, e.g.,
  `t0 = a.[u256 (int)(32 *64u i)];`

Jasmin errors

- Easy case: syntax errors
- Slighly 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;
}
```

Jasmin errors

- Easy case: syntax errors
- Slighly 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

Vectorization in jasmin

- Jasmin supports 128-bit XMM and 256-bit YMM registers: `u128` and `u256`
- Operations through "intrinsics", 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;
}
```

Some current limitations

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;
}
```

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

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, again, where are we?

Efficiency

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

Security

- ???

Correctness

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

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 whishlist**...

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 whishlist**...

Option 3: Avoid variable-time code

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

Secret types

- 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”

Secret types

- 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`

Secret types

- 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)

Secret types

- 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

Secret types

- 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

Summary

- Jasmin is an “easy way to program on assembly level”
  - Easy way to implement conditionals, loops, functions
  - More readable syntax
  - Register allocation
- Guarantees of memory safety, thread safety
- Functional correctness proofs in EasyCrypt
- Constant-time ensured through type system

Jasmin exercise

1. Download https://cryptojedi.org/bkk-school-excercise1.tar.bz2
2. Check that you can build the code:

```
tar xjvf bkk-school-exercise1.tar.bz2
cd bkk-school-exercise1
make
./test
```

3. Make sure that `./test` no longer prints an error message:
   - Implement function `poly1305_verify_jasmin` in `jasmin/poly1305.jazz`
   - See function `poly1305_verify_c` in `c/poly1305.c`
4. Make your implementation pass constant-time check:
   - Check with `jasminc -checkCT jasmin/poly1305.jazz`
   - Hint: The C code is *not* constant time!

Jasmin exercise – Part II

1. Download https://cryptojedi.org/bkk-school-excercise2.tar.bz2
2. Check that you can build the code:

```
tar xjvf bkk-school-exercise2.tar.bz2
cd bkk-school-exercise2
make
./test
```

3. Make sure that `./test` no longer prints an error message:
   - Implement function `gimli_jasmin` in `jasmin/gimli.jazz`
   - See function `gimli_c` in `c/gimli.c`
4. Bonus: Make your Gimli implemetation faster
   - Use `./speed` to see cycle counts