Engineering Cryptographic Software

The Jasmin Framework

Peter Schwabe

January 2026

High-assurance crypto

Reminder:

- Traditionally, crypto software often written in C and assembly.
- Software is very efficient, but neither (guaranteed to be) correct nor (guaranteed to be) secure.

High-assurance crypto

Reminder:

- Traditionally, crypto software often written in C and assembly.
- Software is very efficient, but 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

Reminder:

- Traditionally, crypto software often written in C and assembly.
- Software is very efficient, but 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

Reminder:

- Traditionally, crypto software often written in C and assembly.
- Software is very efficient, but 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

Reminder:

- Traditionally, crypto software often written in C and assembly.
- Software is very efficient, but 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 build **formally verified** crypto software
- Currently three main projects:
  - EasyCrypt proof assistant
  - jasmin programming language
  - Libjade (PQ-)crypto library
- Core team of ≈ 30–40 people
- Discussion forum with >350 people

BOSTON UNIVERSITY

INESCTEC

PQ SHIELD

University of BRISTOL

Inria

Radboud University

CASA
Cyber Security in the Age of Large-Scale Adversaries

ROSENPASS

CryptoExperts

Universidade do Minho

SANDBOXAQ

h_da darmstadt university of applied sciences

MAX PLANCK INSTITUTE FOR SECURITY AND PRIVACY

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

institute IMDEA software

U.PORTO FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

UCLouvain

Formosan black bear

24 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

The **Formosan black bear** (臺灣黑熊, ***Ursus thibetanus formosanus***), also known as the **Taiwanese black bear** or **white-throated bear**, is a subspecies of the Asiatic black bear. It was first described by Robert Swinhoe in 1864. Formosan black bears are endemic to Taiwan. They are also the largest land animals and the only native bears (*Ursidae*) in Taiwan. They are seen to represent the Taiwanese nation.

Because of severe exploitation and habitat degradation in recent decades, populations of wild Formosan black bears have been declining. This species was listed as "endangered" under Taiwan's Wildlife Conservation Act (Traditional Chinese: 野生動物保育法) in 1989. Their geographic distribution is restricted to remote, rugged areas at elevations of 1,000–3,500 metres (3,300–11,500 ft). The estimated number of individuals is 200 to 600.[3]

Formosan black bear

Conservation status

Extinct — Threatened — Least Concern

EX EW CR EN VU NT LC

Vulnerable (IUCN 3.1)[1]

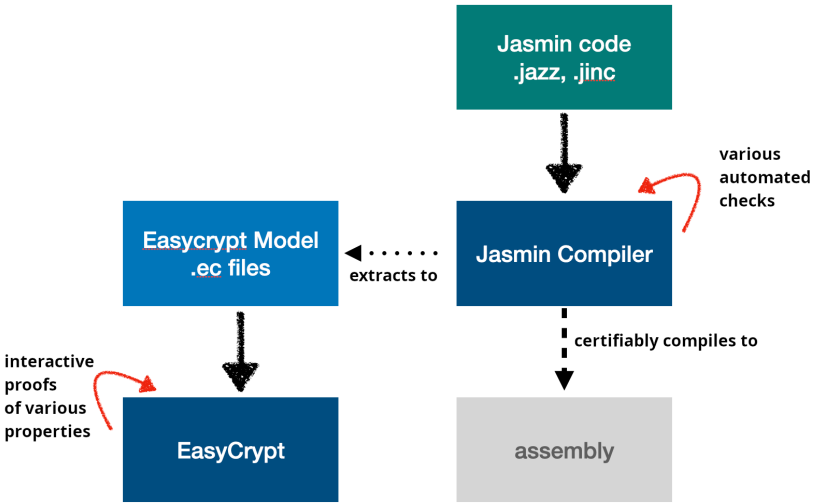
Physical characteristics [edit]

The V-shaped white mark on a bear's chest

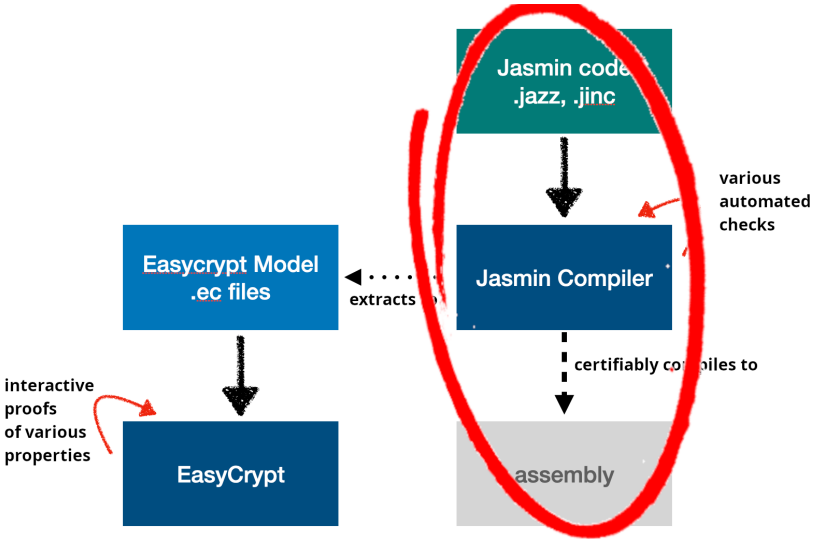
The Formosan black bear is sturdily built and has a round head, short neck, small eyes, and long snout. Its head measures 26–35 cm (10–14 in) in length and 40–60 cm (16–24 in) in circumference. Its ears are 8–12 cm (3.1–4.7 in) long. Its snout resembles a dog's, hence its nickname is "dog bear". Its tail is inconspicuous and short—usually less than 10 cm (3.9 in) long. Its body is well covered with rough, glossy, black hair, which can grow over 10 cm long around the neck. The tip of its chin is white. On the chest, there is a

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

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

- On Nucleo board all the problems we discussed already
- We don't implement `main` in Jasmin
- We don't have I/O in Jasmin

Our first Jasmin program: add 42


```
export fn add42(reg u32 x) -> reg u32 {
  x += 42;
  return x;
}
```

Our first Jasmin program: add 42


```
export fn add42(reg u32 x) -> reg u32 {
  x += 42;
  return x;
}
```

- Save in file, say `add42.jazz`
- Compile with `jasminc -arch arm-m4 add42.jazz -o add42.s`

Our first Jasmin program: add 42


```
export fn add42(reg u32 x) -> reg u32 {
  x += 42;
  return x;
}
```

```
  .thumb
  .syntax unified
  .global add42
  .thumb_func
  .type add42, %function
add42:
  push {lr}
  ADD  r0, r0, #42
  pop  {pc}
  .section  ".note.GNU-stack", "", %progbits
```

- Save in file, say `add42.jazz`
- Compile with `jasminc -arch arm-m4 add42.jazz -o add42.s`

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 `u32`, `u16` etc.
- Jasmin supports arrays of `reg` and `stack` variables:
  - `reg u32[10] a;`
  - `stack u64[100] b;`
- Arrays have **fixed** length (known at compile time)
- 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

Loop examples

`for` loop

```
export fn sum4(reg ptr u32[4] in) -> reg u32 {
  inline int i;
  reg u32 r, t;
  r = 0;
  for i = 0 to 4 {
    t = in[i];
    r += t;
  }
  return r;
}
```

```
sum4:
  push  {lr}
  MOV   r1, #0
  LDR   r2, [r0]
  ADD   r1, r1, r2
  LDR   r2, [r0, #4]
  ADD   r1, r1, r2
  LDR   r2, [r0, #8]
  ADD   r1, r1, r2
  LDR   r2, [r0, #12]
  ADD   r0, r1, r2
  pop   {pc}
```

Loop examples

`while` loop

```
export fn sum4(reg ptr u32[4] in) -> reg u32 {
  reg u32 r, t, i;
  r = 0;
  i = 0;
  while (i < 4) {
    t = in[(uint)i];
    r += t;
    i += 1;
  }
  return r;
}
```

```
sum4:
  push  {lr}
  MOV   r1, #0
  MOV   r2, #0
  b     Lsum4$1
Lsum4$2:
  LDR   r3, [r0, r2, lsl #2]
  ADD   r1, r1, r3
  ADD   r2, r2, #1
Lsum4$1:
  CMP   r2, #4
  bcc   Lsum4$2
  MOV   r0, r1
  pop   {pc}
```

Conditional example

`if-else` statement

```
export fn cond(reg u32 x) -> reg u32 {
  reg u32 r;
  if (x == 42) {
    r = 0;
  }
  else {
    r = 1;
  }
  return r;
}
```

```
cond:
  push  {lr}
  CMP   r0, #42
  beq   Lcond$1
  MOV   r0, #1
  b     Lcond$2
Lcond$1:
  MOV   r0, #0
Lcond$2:
  pop   {pc}
```

Three kinds of “functions”

`export` functions

- Entry points into jasmin-generated code
- Need at least one **`export`** function in a jasmin program
- Follows 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 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 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)

Register allocation and spilling

- Jasmin takes care of register allocation
  - Assign *live* variables to registers
  - Keep track of set of live variables
  - Automatically handle various constraints (e.g., return value in `r0`)

Register allocation and spilling

- Jasmin takes care of register allocation
  - Assign *live* variables to registers
  - Keep track of set of live variables
  - Automatically handle various constraints (e.g., return value in `r0`)
- Jasmin does **not** take care of spilling
  - If there are not enough registers, compilation will fail
  - If constraints cannot be met, compilation will fail
  - Need to manually **spill** to and **unspill** from the stack

Register allocation and spilling

- Jasmin takes care of register allocation
  - Assign *live* variables to registers
  - Keep track of set of live variables
  - Automatically handle various constraints (e.g., return value in `r0`)
- Jasmin does by default **not** take care of spilling
  - If there are not enough registers, compilation will fail
  - If constraints cannot be met, compilation will fail
  - Need to manually **spill** to and **unspill** from the stack
- New Jasmin feature (Dec 18, 2025): **auto-spill**
  - Compile with `-auto-spill-all` will spill/unspill all `reg` variables
  - Can manually mark a variable as `#[nospill]` to prevent this

Register allocation and spilling

- Jasmin takes care of register allocation
  - Assign *live* variables to registers
  - Keep track of set of live variables
  - Automatically handle various constraints (e.g., return value in `r0`)
- Jasmin does by default **not** take care of spilling
  - If there are not enough registers, compilation will fail
  - If constraints cannot be met, compilation will fail
  - Need to manually **spill** to and **unspill** from the stack
- New Jasmin feature (Dec 18, 2025): **auto-spill**
  - Compile with `-auto-spill-all` will spill/unspill all `reg` variables
  - Can manually mark a variable as `#[nospill]` to prevent this
  - Compile with `-auto-spill` will spill/unspill `reg` variables marked as `#[spill]`

Register allocation and spilling

- Jasmin takes care of register allocation
  - Assign *live* variables to registers
  - Keep track of set of live variables
  - Automatically handle various constraints (e.g., return value in `r0`)
- Jasmin does by default **not** take care of spilling
  - If there are not enough registers, compilation will fail
  - If constraints cannot be met, compilation will fail
  - Need to manually **spill** to and **unspill** from the stack
- New Jasmin feature (Dec 18, 2025): **auto-spill**
  - Compile with `-auto-spill-all` will spill/unspill all `reg` variables
  - Can manually mark a variable as `#[nospill]` to prevent this
  - Compile with `-auto-spill` will spill/unspill `reg` variables marked as `#[spill]`
  - No *efficient* spilling, no automated optimization!
  - Also: no automatic spilling for register *arrays*

Examples


```
export fn sum20() -> reg u32 {
  reg u32 t0, t1, t2, t3, t4;
  reg u32 t5, t6, t7, t8, t9;
  reg u32 t10, t11, t12, t13, t14;
  reg u32 t15, t16, t17, t18, t19;
  reg u32 r;

  t0 = 0;
  t1 = 1;
  ...
  t19 = 19;

  r = 0;
  r = r + t0;
  r = r + t1;
  ...
  r = r + t19;

  return r;
}
```

- This will *not* compile without auto-spilling
- This will *not* compile `-auto-spill`
- This *will* compile with `-auto-spill-all`

Examples


```
export fn sum20() -> reg u32 {
  #[spill] reg u32 t0, t1, t2, t3, t4;
  #[spill] reg u32 t5, t6, t7, t8, t9;
  reg u32 t10, t11, t12, t13, t14;
  reg u32 t15, t16, t17, t18, t19;
  reg u32 r;

  t0 = 0;
  t1 = 1;
  ...
  t19 = 19;

  r = 0;
  r = r + t0;
  r = r + t1;
  ...
  r = r + t19;

  return r;
}
```

- This will *not* compile without auto-spilling
- This *will* compile `-auto-spill`
- More efficient than with `-auto-spill-all`

Memory and thread safety

The old way

- Static safety check:
  `jasminc -checksafety`
- Great when it works
- Takes a long time (not modular)
- Often fails for safe code

Memory and thread safety

The old way

- Static safety check:
  `jasminc -checksafety`
- Great when it works
- Takes a long time (not modular)
- Often fails for safe code

The new way

- Master's thesis by **Francisca Barros**
- Modular design
- Safety contracts and assertions in Jasmin
- Automatic discharge of assertions
- Prove remaining assertions in EasyCrypt

```
fn _gen_matrix_avx2
( reg mut ptr u16[MLKEM_K * MLKEM_K * MLKEM_N] matrix
, reg const ptr u8[32] rho
, #spill_to_mmx reg u64 transposed
) -> reg ptr u16[MLKEM_K * MLKEM_K * MLKEM_N]
requires {is_arr_init(rho,0,32) && 0<= transposed && transposed <= 1}
ensures  {is_arr_init(result.0,0,MLKEM_K * MLKEM_K * MLKEM_N * 2)}
{
  ...
}
```

Safety checking for this course

- Assignment framework as `check-safety` target in Makefile:

```
make check-safety
```

- This uses the old safety checker
- Feel free to run and try it but:
  - be prepared for it to fail on safe code
  - be prepared for it to take quite some time

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

- ???

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 has been 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 has been 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

Example


```
export fn cmov256(#[public] reg ptr u32[8] pr px,
                  #[secret] reg u32 b)
                  -> #[public] reg ptr u32[8]
{
  if (b ==  1) {
    pr = #copy(px);
  }
  pr = pr;
  return pr;
}
```

- Check this program with `jasmin-ct --arch arm-m4 FILENAME.jazz`:

  ```
  constant type checker: b has type secret it needs to be public
  ```

Example


```
export fn cmov256(#[public] reg ptr u32[8] pr px,
                  #[secret] reg u32 b)
                  -> #[public] reg ptr u32[8]
{
  () = #declassify(b);
  if (b ==  1) {
    pr = #copy(px);
  }
  pr = pr;
  return pr;
}
```

- This program will pass the constant-time checker

Hints and possible pitfalls

- Start writing (or modifying) a simple `export` function
- Make sure that it compiles, behaves like you expect
- See, e.g., `playground2-jasmin/src/myjasmin.jazz`
- Then move to `assignment0-sum`

Hints and possible pitfalls

Careful with pointer arguments, there are two kinds:

- Pointers to memory of fixed length (treated like arrays):
  - Argument type, e.g., `reg ptr u32[8] x`
  - Load second element into `reg u32 a`:

```
a = x[1];
```

- Pointers to memory of variable length
  - Argument type is `reg u32 p`, typically second argument `reg u32 plen`
  - Load second 4-byte word into `reg u32 a`:

```
a = (32u)[p + 4];
```

Hints and possible pitfalls

Jasmin requires explicit casts, syntax is *unexpected*:

```
reg u32 a;
reg u8 b;
a = 42;
b = (8u)a;
b += 23;
a = (32u)b;
```

Jasmin resources

Jasmin documentation:

https://jasmin-lang.readthedocs.io/en/stable/

Examples of Jasmin Cortex-M4 code:

https://github.com/jasmin-lang/jasmin/tree/main/compiler/tests/success/arm-m4

Chapter 4 of Ph.D. thesis by Tiago Oliveira (focused on x86_64):

https://repositorio-aberto.up.pt/bitstream/10216/144015/2/580364.pdf

Formosa Crypto Zulipchat:

https://formosa-crypto.zulipchat.com