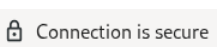
High-Assurance and High-Speed Cryptography with Jasmin and EasyCrypt

Jean-Christophe Léchenet

Centre Inria d'Université Côte d'Azur

Summer school "Proof assistants and applications" – September 3rd, 2026

Cryptography

- Definition: "Cryptography, or cryptology, is the practice and study of techniques for secure communication in the presence of adversarial behavior."[1]
- Goal: ensure some properties
  - confidentiality
  - authentication
- Applications
  - Web 

  - Messaging 

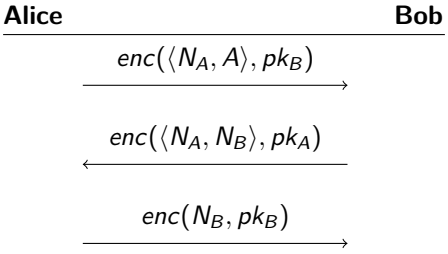
  - Blockchain 

[1] https://en.wikipedia.org/wiki/Cryptography

Protocols based on primitives

- Protocol
  - Actors trying to communicate following a given scheme
  - Some attackers may listen/write
  - Examples: TLS, SSH
- Primitive
  - Building block that is assumed hard to break
  - Examples: hash function (SHA-1), symmetric encryption (AES)

Example: Needham–Schroeder protocol



$dec(enc(t, pk_X), sk_X) = t$

Several sources of insecurity

- Broken protocol
  - Needham-Schroeder published in 1978
  - Man-in-the-middle attack in 1995
- Broken primitive
  - SHA-1
    - theoretical collision in 2005
    - deprecated in 2011
    - practical collision in 2017 (SHAttered)
- Broken implementations
  - A lot of examples!

Predictable private keys

- In OpenSSL
- Introduced in 2006, discovered in 2008
- Valgrind complaining about some lines
- Commented out, but actually broke randomness

Sony PS3 vulnerability

- PS3 released in 2006, discovered in 2010
- PS3 firmware signed with ECDSA
- A nonce was actually a constant
- Private key recovered

Heartbleed

- In OpenSSL
- Introduced in 2012, discovered in 2014
- Heartbeat request: payload + length
- No check that the payload has the right length

goto fail

- In Apple's SSL/TLS library
- Introduced in 2012, discovered in 2014

```
static OSStatus SSLVerifySignedServerKeyExchange(SSLContext* ctx, bool isRsa, SSLBuffer
signedParams, uint8_t* signature, uint16_t signatureLen) {
    OSStatus err;
    // ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

    // ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

[1] https://en.wikipedia.org/wiki/Unreachable_code#goto_fail_bug

Still in 2026

LINK	CVE-2026-10097	High	wolfSSL's AVX2-optimized ML-KEM implementation (mlkem_cmp_avx2) compares only 1536 of the 1568 ciphertext bytes during the Fujisaki-Okamoto re-encryption check in ML-KEM-1024 decapsulation. Ciphertexts that differ from the expected re-encryption solely in bytes 1536-1567 bypass implicit rejection and are accepted as valid, breaking IND-CCA2 security. An attacker able to submit chosen ciphertexts to a decapsulation oracle that uses a static ML-KEM-1024 key, and to observe whether the genuine shared secret or the implicit-rejection secret was produced, can use this as a plaintext-checking oracle to recover the private key. A proof of concept recovered a full ML-KEM-1024 private key with approximately 98% success using roughly 350 chosen ciphertexts. The flaw is a deterministic logic error and does not rely on timing measurements. Fixed in PR 10430.	27 days	5.9.2

Side-channel attacks

- Attacks based on extra observations (e.g. timing, power consumption)
- Against timing attacks: cryptographic constant time discipline
  - branch conditions cannot depend on secrets
  - memory accesses cannot depend on secrets

Spectre attacks

- Discovered in 2017
- Based on speculation

Cryptography: a good fit for formal methods

- critical
- same code running on billions of devices
- open source
- clear specifications (RFCs, mathematics)
- a lot of bugs discovered in the past
- avoid a "crisis of rigor"
  "In our opinion, many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor."[1]

[1] Bellare and Rogaway, 2006

Several sources of insecurity

- Broken protocol
  - Answer: dedicated tools (e.g. ProVerif, CryptoVerif, Squirrel Prover, Tamarin)
  - Not this talk
- Broken primitive
  - Answer: security proofs in EasyCrypt
- Broken implementations
  - Answer: implementations proved correct in Jasmin

Outline

EasyCrypt

- Proof assistant specialized to proofs in cryptography
- Probabilistic programs expressed in pWhile
- Several logics: in particular, pRHL
- Game-based proofs
- Goal: bound the advantage of the attacker
- CRYPTO 2011: Computer-Aided Security Proofs for the Working Cryptographer (Barthe, Grégoire, Heraud, Zanella-Béguelin)
  - Best paper award: recognized by cryptographers

Outline

Cryptographic code uses assembly

- for performance reasons
- E.g. OpenSSL include assembly files or perlASM files
- Still in 2026!

Commit 96b16b9

dannytsen authored and **t8m** committed 4 days ago · ✓ 127 / 133

```
ppc64le: Optimized MLDSA NTT, supports p8 and above architectures.

1. rebased.
2. fixed tab.

Optimized MLDSA NTT implementation for ppc64le (p8+).

Supporting files include,
ppc64/mldsa_ntt_ppc_4x.S: supports ossl_ml_dsa_poly_ntt.
ppc64/mldsa_intt_ppc_4x.S: supports ossl_ml_dsa_poly_ntt_inverse.
ppc64/mldsa_poly_ntt_mul.S: supports ossl_poly_ntt_mult.
ppc64/mldsa_poly_add.S: supports poly_add and poly_sub.

Modified build.info to support ppc64le assembly implementation.
Added new definitions of MLDSA_NTT_ASM and MLDSA_POLY_ADD_ASM for
NTT, inverse NTT and poly_add and poly_sub for optimized assembly
implementation.

This is the initial archtecture specific implementation so can be mdified
to adapt to a new build structures.

Baseline speed test:

                    keygen      signs     verify keygens/s    sign/s  verify/s
         ML-DSA-44 0.000141s 0.000932s 0.000171s    7115.2    1073.3    5860.9
         ML-DSA-65 0.000238s 0.001471s 0.000255s    4207.2     679.6    3920.2
         ML-DSA-87 0.000340s 0.001719s 0.000387s    2939.4     581.8    2584.8

Optimized:
         ML-DSA-44 0.000072s 0.000253s 0.000060s   13866.8    3954.7   16735.7
         ML-DSA-65 0.000138s 0.000400s 0.000095s    7253.2    2500.2   10532.4
         ML-DSA-87 0.000187s 0.000485s 0.000159s    5338.5    2061.4    6269.8

The optimized code runs around 3.6 times faster than the original C implementation.

Signed-off-by: Danny Tsen <dtsen@us.ibm.com>
Reviewed-by: Shane Lontis <shane.lontis@oracle.com>
Reviewed-by: Tomas Mraz <tomas@openssl.foundation>
Merge-date: Thu Aug 27 16:26:36 2026
Merged-from: #29611
```

Assembly is not perfect

- Painful to write and maintain
- Difficult to reason on
- Answer: a dedicated language, Jasmin

Goal: high-speed and high-assurance cryptographic primitives

- high-assurance?
  - safety and security
  - with formal guarantees
- high-speed?
  - no concession on speed
  - the users must be able to write what they want to write

Jasmin, a language between C and assembly

- A syntax *à la* C/Rust
- But with all the hardware constraints

Jasmin, high- or low-level language?

- High-level
  - Structure: functions, loops
  - Variables, arrays
- Low-level
  - Intrinsics
  - Flags
  - Variables are either `reg` or `stack`
  - Sizes must be explicitly given to integer variables
  - Architecture-specific code

"Hello world!" (on x86-64)

```
1 export fn zero() -> reg u64 {
2     reg u64 res = 0;
3     return res;
4 }
```

"Hello world!" (on x86-64)

```
1 export fn id(reg u64 x) −→ reg u64 {
2
3     return x;
4 }
```

✗

"Hello world!" (on x86-64)

```
1 export fn id(reg u64 x) −→ reg u64 {
2     reg u64 y = x;
3     return y;
4 }
```

A more realistic example

```
 1 inline fn gimli_body(reg u128 x, reg u128 y, reg u128 z) ⟶ reg u128, reg u128, reg u128 {
 2     reg u128 a b c d e;
 3     inline int round;
 4     for round = 24 downto 0 {
 5         x = rotate(x, 24);
 6         y = rotate(y, 9);
 7         a = shift(z, 1);
 8         b = y & z;
 9         c = shift(b, 2);
10         d = x ^ a;
11         e = d ^ c;
12         a = x | z;
13         b = shift(a, 1);
14         c = y ^ x;
15         d = c ^ b;
16         a = x & y;
17         b = shift(a, 3);
18         c = z ^ y;
19         x = c ^ b;
20         y = d;
21         z = e;
22         if round % 4 == 0 { x = #VPSHUFD(x, (4u2)[2, 3, 0, 1]); }
23         if round % 4 == 2 { x = #VPSHUFD(x, (4u2)[1, 0, 3, 2]); }
24         if round % 4 == 0 { reg u32 m = 0x9e377900 +32u round; a = (128u) m; x ^= a; }
25     }
26     return x, y, z;
27 }
```

Jasmin is also a compiler

- Compiler from Jasmin to assembly
  - x86_64
  - ARMv7 32 bits
  - RISC-V IM 32 bits
- A strange compiler
  - Not very optimizing (on purpose)
  - Not very accommodating: a lot of programs are rejected

Jasmin is also a *verified* compiler

- A compiler, together with a proof of correctness
- Intuitively: behaviour is preserved
- Formally:
  - Give a semantics to the source and target languages
  - Theorem: if a source program S is safe and is successfully compiled into target program T, then any behaviour of T is a behaviour of S
  - Corollary: what is proved about the behaviours of S applies also to T
  - What we rather prove: if S is successfully compiled into T, then any safe behaviour of S is a behaviour of T
- Other famous verified compilers: CompCert (for C), CakeML (for ML)

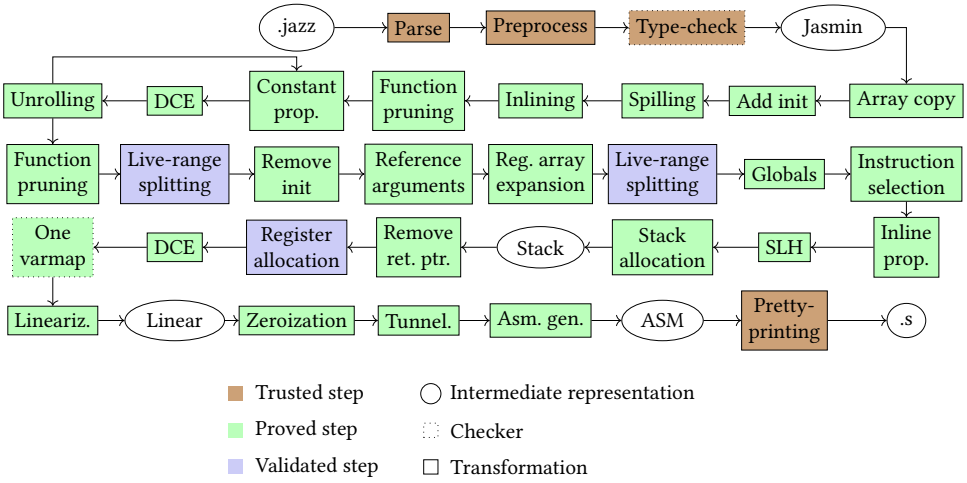
Correctness theorem

```
Theorem alloc_progP nrip nrsp data oracle_g oracle (P: uprog) (SP: sprog) fn:
  alloc_prog nrip nrsp data oracle_g oracle P = ok SP → (* compilation succeeds *)
  forall ev scs1 m1 vargs1 scs1' m1' vres1,
  sem_call P ev scs1 m1 fn vargs1 scs1' m1' vres1 → (* function fn has some semantics in the source *)
  forall rip m2 vargs2,
  extend_mem m1 m2 rip data → (* target mem m2 contains source mem m1 and global data *)
  wf_args (Z.of_nat (size data)) rip m1 m2
    (map (omap pp_writable) (oracle fn).(sao_params))
    (map (oapp pp_align U8) (oracle fn).(sao_params)) vargs1 vargs2 → (* memory slots for reg ptr args *)
  Forall3 (value_eq_or_in_mem m2) (oracle fn).(sao_params) vargs1 vargs2 → (* arg values coincide *)
  alloc_ok SP fn m2 → (* enough space in the stack *)
  exists m2' vres2, [∧
    sem_call SP rip scs1 m2 fn vargs2 scs1' m2' vres2, (* function fn has some semantics in the target *)
    extend_mem m1' m2' rip data, (* target mem m2' contains source mem m1' and global data *)
    Forall3 (wf_result vargs1 vargs2) (oracle fn).(sao_return) vres1 vres2, (* reg ptr res were in args *)
    Forall3 (value_eq_or_in_mem m2') (oracle fn).(sao_return) vres1 vres2 & (* res values coincide *)
    mem_unchanged_params m1 m2 m2'
      (map (omap pp_writable) (oracle fn).(sao_params)) vargs1 vargs2]. (* rest of the stack is untouched *)
```

Jasmin compiler in practice

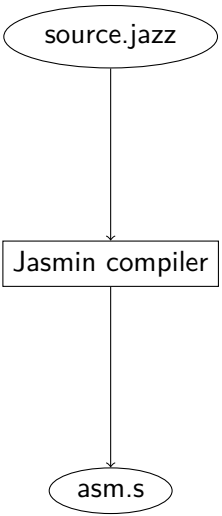
- A mix of Rocq code and OCaml code
- The Rocq part is extracted into OCaml, then compiled

Jasmin compiler: compilation passes



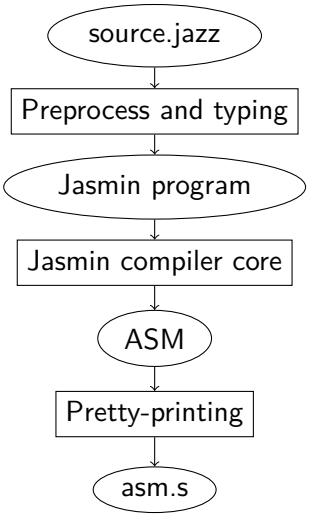
(courtesy of Vincent Laporte)

An ecosystem



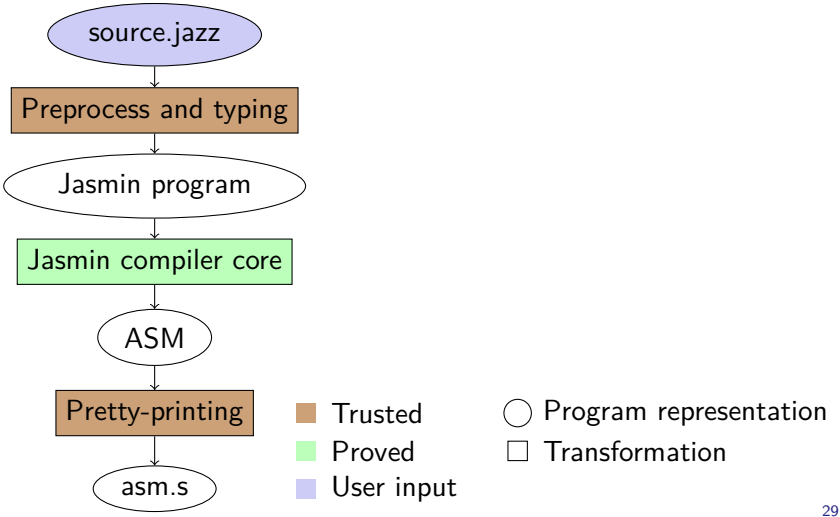
◯ Program representation
□ Transformation

An ecosystem

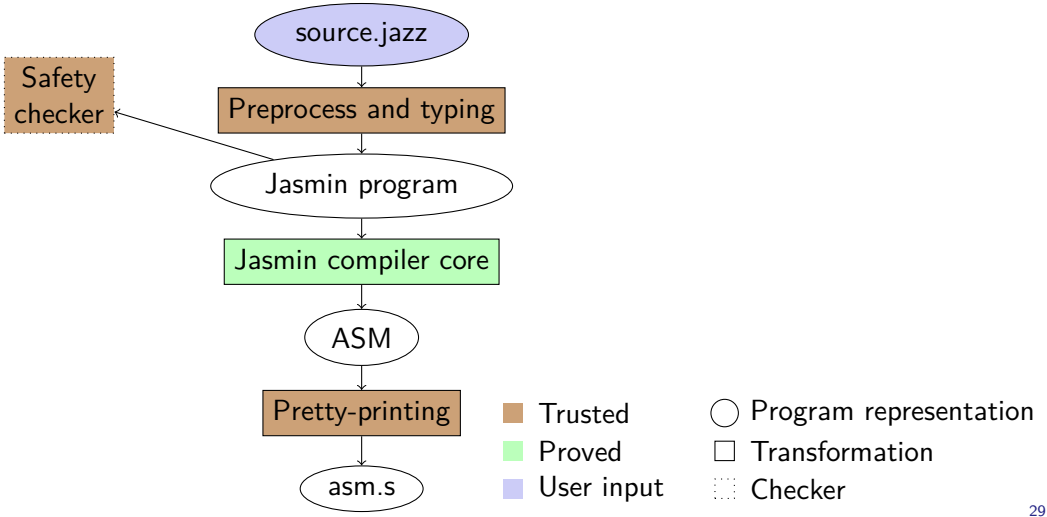


◯ Program representation
□ Transformation

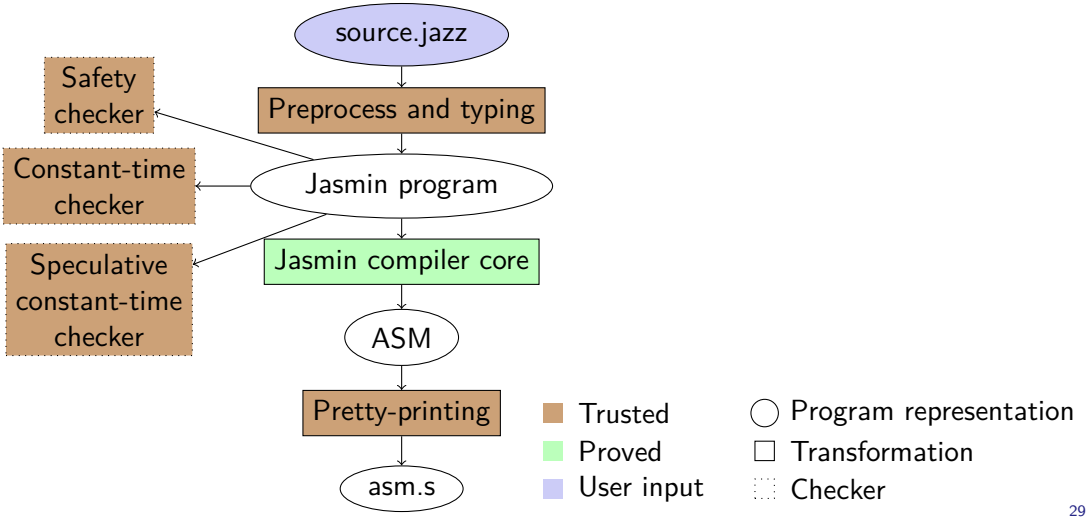
An ecosystem



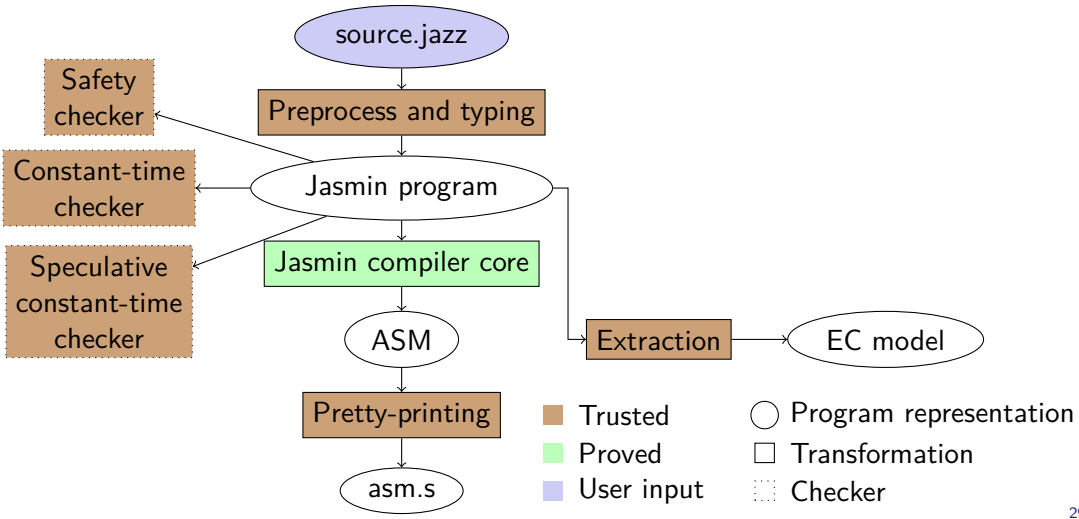
An ecosystem



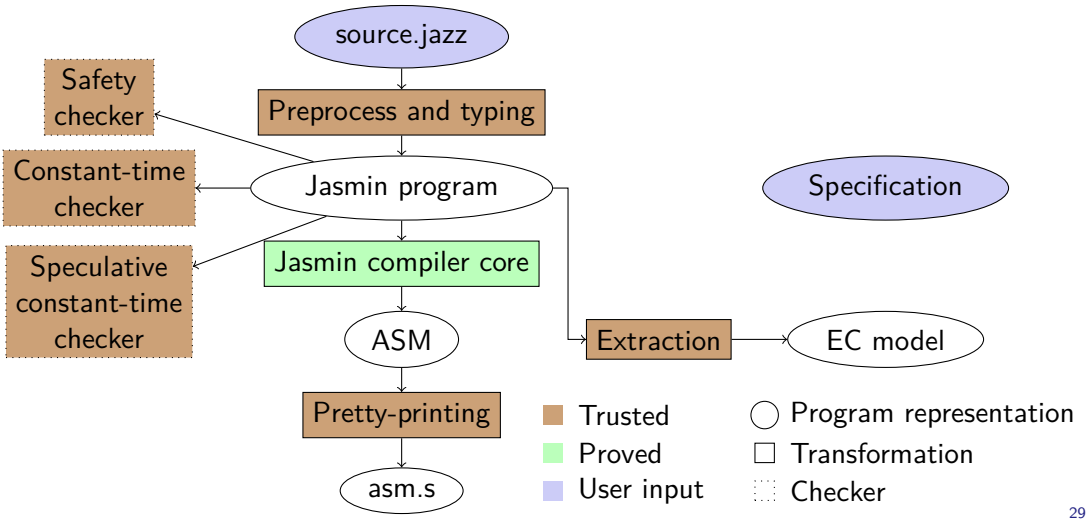
An ecosystem



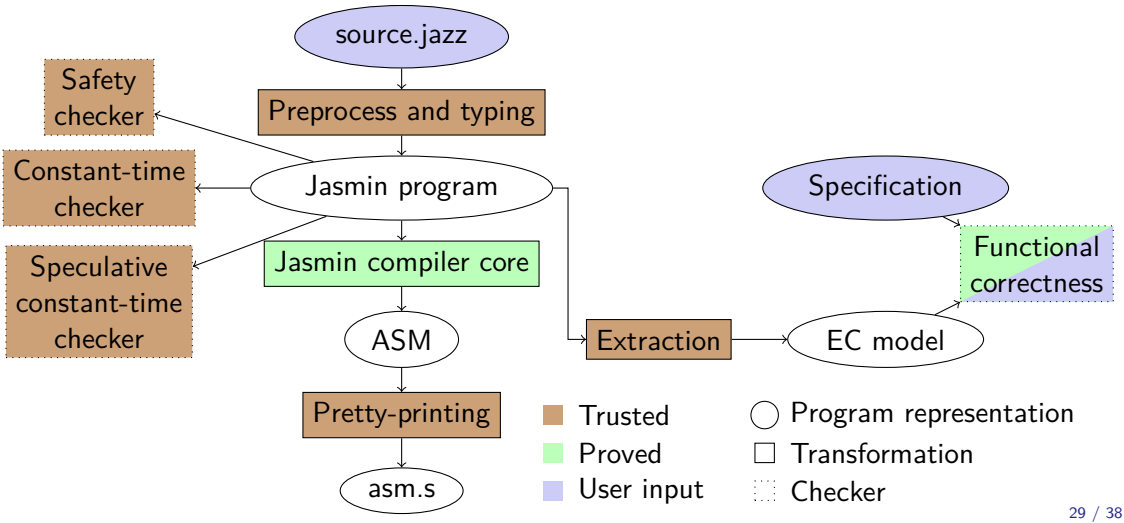
An ecosystem



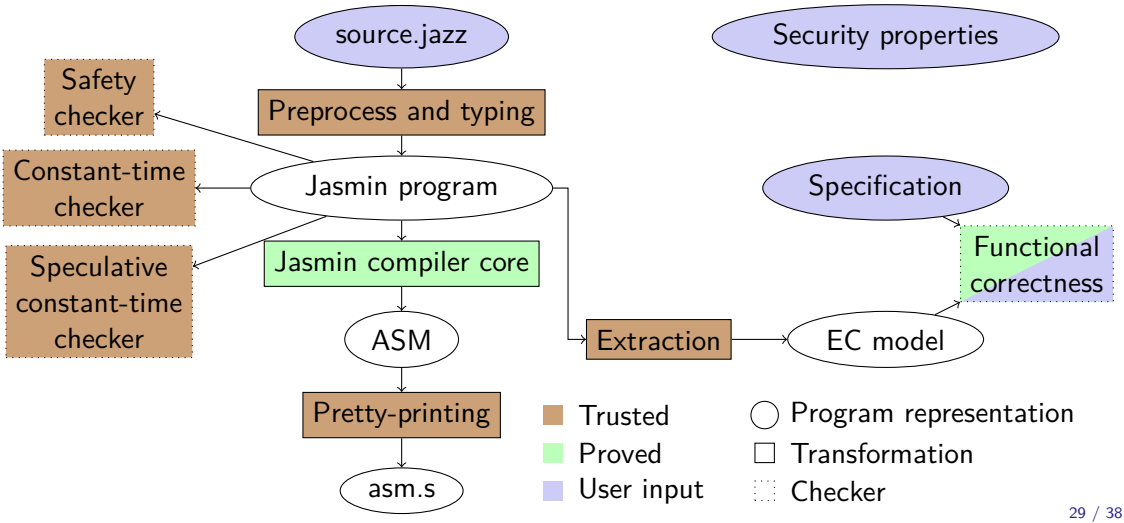
An ecosystem



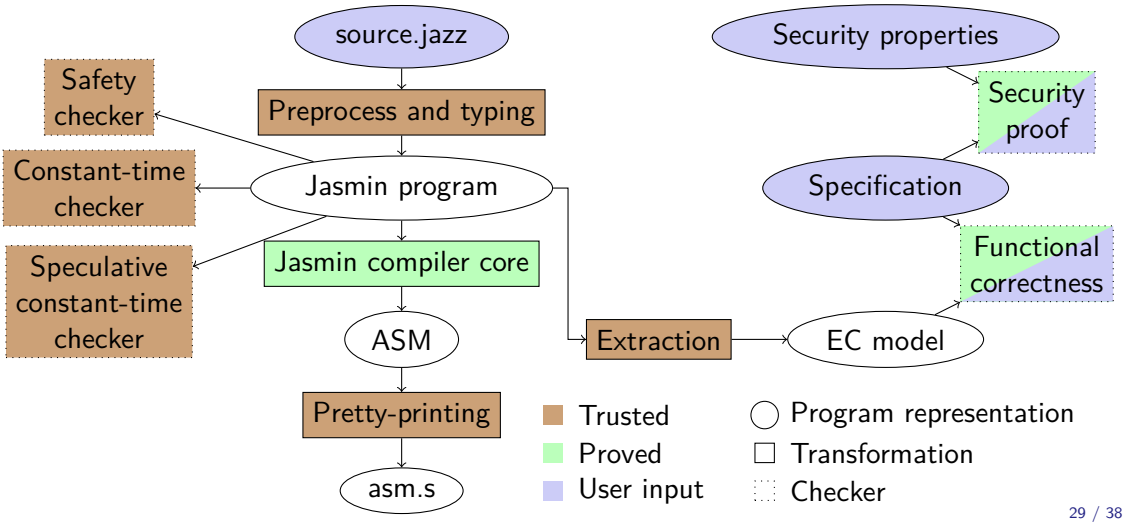
An ecosystem



An ecosystem



An ecosystem



Formosa Crypto, a (mostly) European collaboration

- From November 2021
- `https://formosa-crypto.org/`

Goal: libjade, a cryptographic library

- Implementations written in Jasmin
- Proved safe
- Proved correct w.r.t. a specification
- Proved CT and SCT

Free software!

- Jasmin, `https://github.com/jasmin-lang/jasmin`, MIT
- EasyCrypt, `https://github.com/EasyCrypt/easycrypt`, MIT
- libjade, `https://github.com/formosa-crypto/libjade`, CC0

Outline

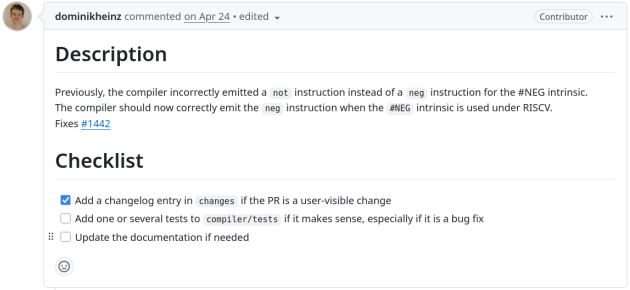
What is really proved?

- TCB: Trusted Computed Base
  - tools: Rocq, EasyCrypt
  - unverified parts: parsing, extraction from Jasmin to EasyCrypt
  - specification: modeling of assembly languages

[1] https://www.galois.com/articles/what-works-and-doesnt-selling-formal-methods

What is really proved?

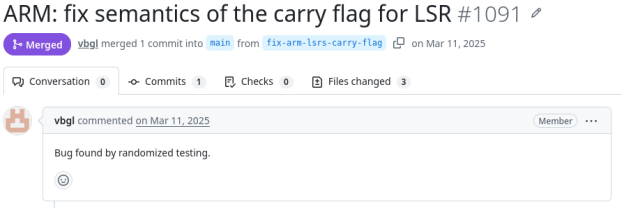
- TCB: Trusted Computed Base
  - tools: Rocq, EasyCrypt
  - **unverified parts: parsing, extraction from Jasmin to EasyCrypt**
  - specification: modeling of assembly languages



[1] https://www.galois.com/articles/what-works-and-doesnt-selling-formal-methods

What is really proved?

- TCB: Trusted Computed Base
  - tools: Rocq, EasyCrypt
  - unverified parts: parsing, extraction from Jasmin to EasyCrypt
  - **specification: modeling of assembly languages**



[1] https://www.galois.com/articles/what-works-and-doesnt-selling-formal-methods

Writing proofs is slow

- Better prove a working feature!
- Sometimes too costly
  - The proof that the compiler preserves CT was not merged

Software engineering -> proof engineering

- Same challenges as software engineering, but worse
- External dependencies (Rocq, MathComp)
- Internal dependencies (Jasmin, EasyCrypt, Jasmin implementations, EasyCrypt proofs)

One success

- Kyber/ML-KEM post-quantum scheme
- Highly-optimized AVX2 implementation
- Proved safe, CT, SCT
- Proved IND-CCA secure
- Integrated in Signal's backend infrastructure  **Signal**

Thank you!