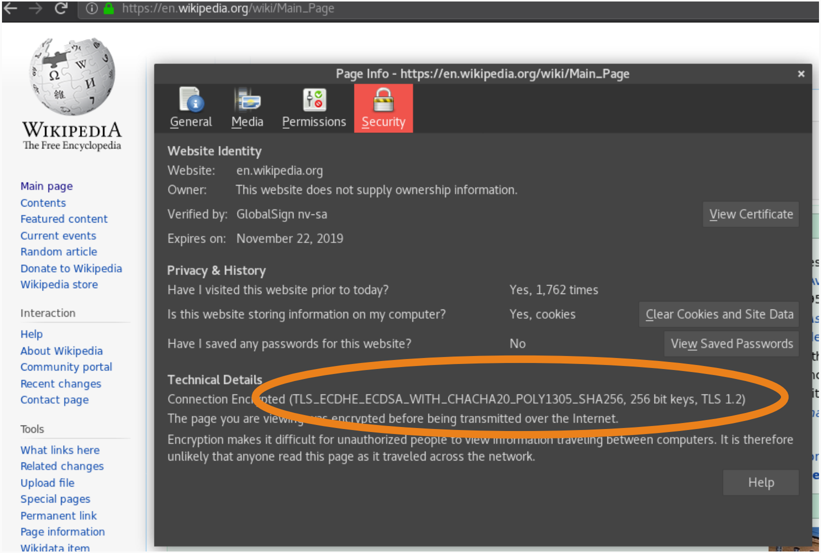
High-Assurance and High-Speed Cryptographic Implementations Using the Jasmin Language

---

J.B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, **A. Koutsos**, V. Laporte, T. Oliveira, P-Y. Strub

Octobre 9th, 2019

Context



Page Info - https://en.wikipedia.org/wiki/Main_Page

Website Identity

Website: en.wikipedia.org
Owner: This website does not supply ownership information.
Verified by: GlobalSign nv-sa
Expires on: November 22, 2019

Privacy & History

Have I visited this website prior to today?	Yes, 1,762 times
Is this website storing information on my computer?	Yes, cookies
Have I saved any passwords for this website?	No

Technical Details

Connection Encrypted (TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256, 256 bit keys, TLS 1.2)
The page you are viewing was encrypted before being transmitted over the Internet.

Encryption makes it difficult for unauthorized people to view information traveling between computers. It is therefore unlikely that anyone read this page as it traveled across the network.

Context

Cryptographic Libraries

Developing cryptographic libraries is hard, as the code must be:

- **efficient:** pervasive usage, on large amount of data.
- **functionally correct:** the specification must be respected.
- **protected against side-channel attacks:** constant-time implementation.

Context

Side-Channel Attacks

Exploit **auxilliary information** to break a cryptographic primitive.

Context

Side-Channel Attacks

Exploit **auxilliary information** to break a cryptographic primitive.

Constant-Time Programming

- Countermeasure against **timing** and **cache** attacks.
- **Control-flow** and **memory accesses** should not depend on **secret** data.
- Crypto implementations without this property are vulnerable.

Difficulties

Constraints

- Efficiency: **low-level** operations and **vectorized** instructions.
- Functional Correctness: **readable** code, with **high-level abstractions**.
- Side-Channel Attacks Protection: **control** over the executed code.

Gap Between Source and Assembly

Source

- **High-level** abstractions.
- **Readable** code.

Gap Between Source and Assembly

Source

- **High-level** abstractions.
- **Readable** code.

Source is not Security/Efficiency Friendly

- Trust compiler (GCC or Clang).
- Certified compilers are less efficient (CompCert).
- Optimizing can break side channel resistance.

Preservation of Constant-Timeness?

Before

```
int cmove(int x, int y, bool b) {
  return x + (y-x) * b;
}
```

Preservation of Constant-Timeness?

Before

```
int cmove(int x, int y, bool b) {
  return x + (y-x) ∗ b;
}
```

After

```
int cmove(int x, int y, bool b) {
  if (b) {
    return y;
  } else {
    return x;
  }
}
```

Gap Between Source and Assembly

Assembly

- **Efficient** code.
- **Control** over the program execution.

Gap Between Source and Assembly

Assembly

- **Efficient** code.
- **Control** over the program execution.

Assembly is not Programmer/Verifier Friendly

- The code is obfuscated.
- More error prone.
- Harder to prove/analyze.

Jasmin: High Assurance Cryptographic Implementations

Fast and Formally Verified Assembly Code

- **Source language**: assembly in the head with formal semantics $\Longrightarrow$ programmer & verification friendly
- **Compiler**: predictable & formally verified (in Coq) $\Longrightarrow$ programmer has control and no compiler security bug
- **Verification tool-chain**:
  - Functional correctness.
  - Side-channel resistance (constant-time).
  - Safety.

Implementations in Jasmin

TLS 1.3 components : ChaCha20, Poly1305, Curve25519.

The Jasmin Language

Initialization of ChaCha20 State

```
inline fn init(reg u64 key nonce, reg u32 counter) → stack u32[16] {
  inline int i;
  stack u32[16] st;
  reg u32[8] k;
  reg u32[3] n;

  st[0] = 0x61707865;
  st[1] = 0x3320646e;
  st[2] = 0x79622d32;
  st[3] = 0x6b206574;

  for i=0 to 8 {
    k[i] = (u32)[key + 4*i];
    st[4+i] = k[i];
  }

  st[12] = counter;

  for i=0 to 3 {
    n[i] = (u32)[nonce + 4*i];
    st[13+i] = n[i];
  }

  return st;
}
```

Zero-Cost Abstractions

- Variable names.
- Arrays.
- Loops.
- Inline functions.

User Control: Loop Unrolling

```
for i=0 to 15 {
  k[i] = st[i];
}
```

```
while(i < 15) {
  k[i] = st[i];  i += 1;
}
```

For Loops

- Fully unrolled.
- The value of the counter is propagated.
- The source code still readable and compact.

While Loops

- Untouched.

User Control: Register or Stack

- Jasmin has three kinds of variables:
  - register variables (reg).
  - stack variables (stack).
  - global variables (global).
- Arrays can be register arrays or stack arrays.
- Spilling is done manually (by the user).

```
inline fn sum_states(reg u32[16] k, stack u32 k15, stack u32[16] st) → reg u32[16], stack u32
{
  inline int i;
  stack u32 k14;

  for i=0 to 15 {
    k[i] += st[i];
  }

  k14    = k[14];   k[15]  = k15;     // Spilling
  k[15] += st[15];
  k15    = k[15];   k[14]  = k14;     // Spilling

  return k, k15;
}
```

User Control: Instruction-Set

- Direct memory access.

```
reg u64 output, plain;

for i=0 to 12 {
  k[i] = (u32)[plain + 4*i];
  (u32)[output + 4*i] = k[i]; }
```

- The carry flag is an ordinary boolean variable.

```
reg u64[3] h;
reg bool cf0 cf1;
reg u64 h2rx4 h2r;

h2r       += h2rx4;
cf0, h[0] += h2r;
cf1, h[1] += 0 + cf0;
_  , h[2] += 0 + cf1;
```

User Control : Instruction-Set

- Most assembly instructions are available.

```
of, cf ,sf, pf, zf, z = x86_ADC(x, y, cf);

of, cf, x = x86_ROL_32(x, bits);
```

- Vectorized instructions (SIMD).

```
k[0] +8u32= k[1];  // vectorized addition of 8 32-bits words;

k[1] = x86_VPSHUFD_256(k[1], (4u2)[0,3,2,1]);
```

The Jasmin Compiler

The Compiler

Goals And Features

- Predictability and control of generated assembly.
- Preserves semantics (machine-checked in Coq).
- Preserves side-channel resistance

Compilation

Passes and Optimizations

- For loop unrolling.
- Function inlining.
- Constant-propagation.
- Sharing of stack variables.
- Register array expansion.
- Lowering.
- Register allocation.
- Linearisation.
- Assembly generation.

Semantic Preservation

Compilation Theorem (Coq)

$$\begin{aligned}&\forall p, p'.\ \mathrm{compile}(p) = \mathrm{ok}(p') \Rightarrow\\ &\forall v_a, m, v_r, m'.\mathrm{enough\text{-}stack\text{-}space}(p', m) \Rightarrow\\ &v_a, m \Downarrow^{p} v_r, m' \ \Rightarrow\ v_a, m \Downarrow^{p'} v_r, m'\end{aligned}$$

Remarks

- The compiler uses validation.
- We may need some extra memory space for p':

$$\mathrm{enough\text{-}stack\text{-}space}(p', m)$$

- If p is not safe, i.e. $v_a, m \Downarrow^{p} \perp$, then we have no guarantees.

Functional Correctness

Functional Correctness

Methodology

- We start from a **readable reference implementation**:
  - Using a mathematical specification (e.g. in $\mathbb{Z}/p\mathbb{Z}$).
  - Or a simple imperative specifications.
- We gradually transform the **reference implem.** into an **optimized implem.**:
  - We prove that each transformation **preserves functional correctness** by equivalence (game-hoping).
- We prove additional properties of the final implementation:
  - **Constant-time** by program equivalence.
  - **Safety** by static analysis.

Functional Correctness

Gradual Transformation

We perform functional correctness proofs by game hopping:

$$c_{\mathrm{ref}} \sim c_1 \sim \ldots \sim c_n \sim c_{\mathrm{opt}}$$

EasyCrypt

- Jasmin programs are translated into EasyCrypt programs.
- EasyCrypt model for Jasmin (memory model + instructions).
- Equivalences are proved in EasyCrypt.

Functional Correctness

Relational Hoare Logic

A judgment $\{P\}\ c_1 \sim c_2\ \{Q\}$ is valid if:

$$(m_1, m_2) \in P \;\Rightarrow\; m_1 \Downarrow^{c_1} m_1' \;\Rightarrow\; m_2 \Downarrow^{c_2} m_2' \;\Rightarrow\; (m_1', m_2') \in Q$$

Relational Hoare Logic is provided in EasyCrypt.

Example

- c_1 is the reference implementation (the specification)
- c_2 is the optimized implementation

$$\{\mathrm{args}\langle m_1\rangle = \mathrm{args}\langle m_2\rangle\}\ c_1 \sim c_2\ \{\mathrm{res}\langle m_1\rangle = \mathrm{res}\langle m_2\rangle\}$$

Example: ChaCha20

Stream cipher that iterates a *body* on all the blocks of a message.

Reference

```
while (i < len) {
  chacha_body;
  i += 1;
}
```

Loop tiling

```
while (i + 4 ≤ len) {
  chacha_body;
  chacha_body;
  chacha_body;
  chacha_body;
  i += 4;
}
chacha_end
```

Scheduling

```
while (i + 4 ≤ len) {
  chacha_body4_swapped;
  i += 4;
}
chacha_end
```

Vectorization

```
while (i + 4 ≤ len) {
  chacha_body4_vectorized;
  i += 4;
}
chacha_end
```

Safety

Safety

Definition

A program p is safe under precondition ϕ if and only if:

$$\forall (v, m) \in \phi.\ v, m \not\Downarrow^{p} \bot$$

Why do we Need Safety?

- If p is safe, its execution never crashes.
- The compilation theorem gives no guarantees if p is not safe.
- Jasmin semantics in Easycrypt assumes that p is safe.

Safety

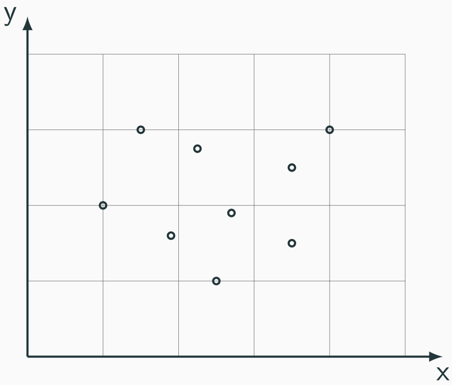
Properties to Check

- Division by zero.
- Variable and array initialization.
- Out-of-bound array access.
- Termination.
- Valid memory access.

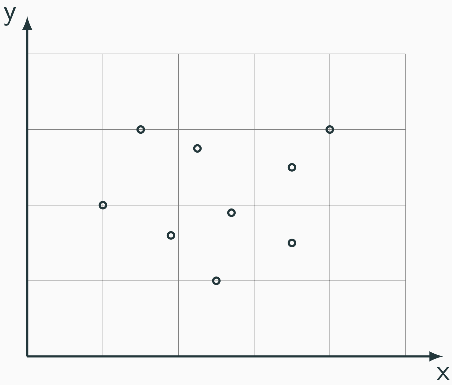
Jasmin

Safety is checked automatically by **static analysis**.

Abstract Interpretation: Abstract Values



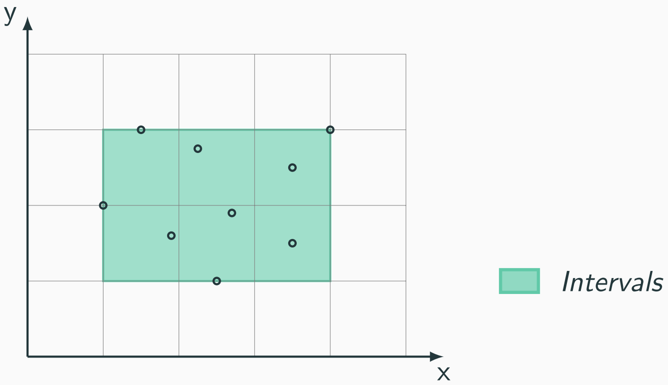
Abstract Interpretation: Abstract Values



Soundness

$X^\sharp$ over-approximates X if and only if $X \subseteq \gamma(X^\sharp)$

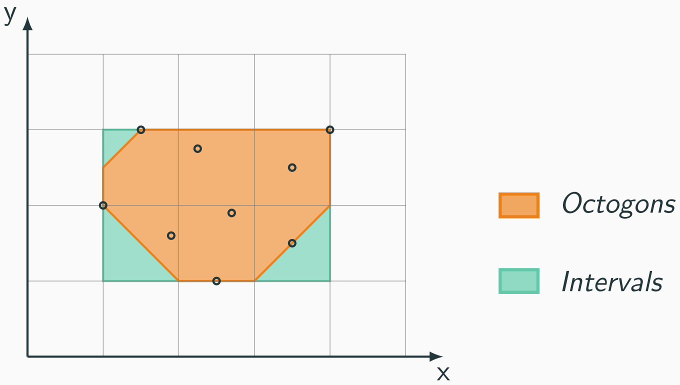
Abstract Interpretation: Abstract Values



Soundness

$X^\sharp$ over-approximates X if and only if $X \subseteq \gamma(X^\sharp)$

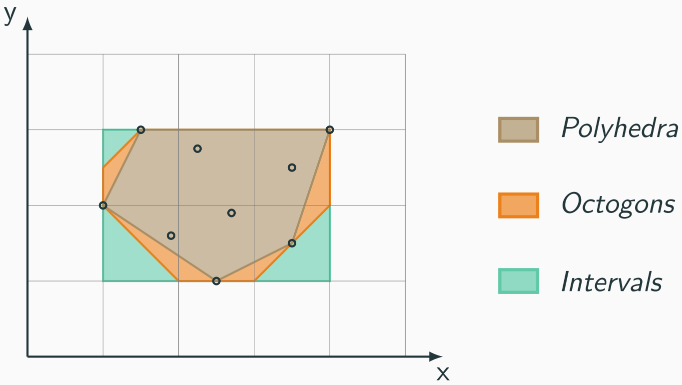
Abstract Interpretation: Abstract Values



Soundness

$X^\sharp$ over-approximates X if and only if $X \subseteq \gamma(X^\sharp)$

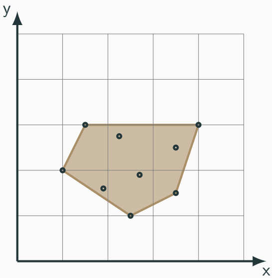
Abstract Interpretation: Abstract Values



Soundness

$X^\sharp$ over-approximates X if and only if $X \subseteq \gamma(X^\sharp)$

Abstract Interpretation: Abstract Transformers

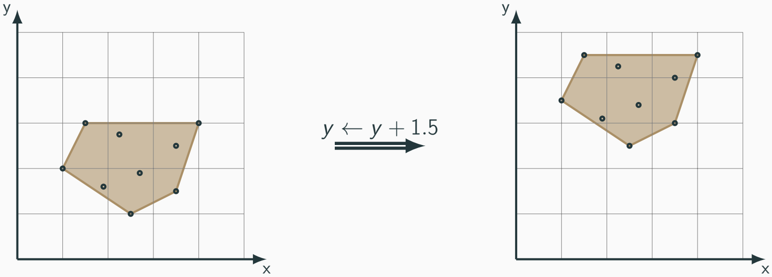


Soundness

$f^\sharp$ over-approximates f if and only if:

$$\forall X^\sharp.\ f \circ \gamma(X^\sharp) \subseteq \gamma \circ f^\sharp(X^\sharp)$$

Abstract Interpretation: Abstract Transformers

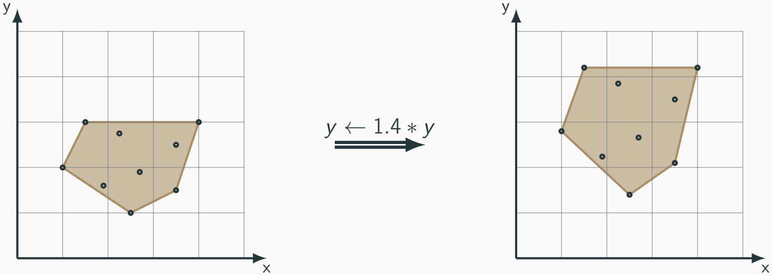


Soundness

f$^\sharp$ over-approximates f if and only if:

$$\forall X^\sharp.\ f \circ \gamma(X^\sharp) \subseteq \gamma \circ f^\sharp(X^\sharp)$$

Abstract Interpretation: Abstract Transformers



$$\xRightarrow{y \leftarrow 1.4 * y}$$

Soundness

f$^\sharp$ over-approximates f if and only if:

$$\forall X^\sharp.\ f \circ \gamma(X^\sharp) \subseteq \gamma \circ f^\sharp(X^\sharp)$$

Safety

Features of the Language

Jasmin is a simple language for static analysis:

- No recursion.
- Arrays size are statically known.
- No dynamic memory allocation.

Example

```
fn load(reg u64 in, reg u64 len) {
 inline int i;
 reg u8 tmp;

 tmp = 0;
 while (len >= 16) {
  for i = 0 to 16 {
   tmp = (u8)[in + i]; }
  in += 16;
  len -= 16; }

 for i = 0 to 16 {
  if i < len {
   tmp = (u8)[in + i]; }}

 return tmp;
}
```

Example

```
fn load(reg u64 in, reg u64 len) {
  inline int i;
  reg u8 tmp;

  tmp = 0;
  while (len >= 16) {
    for i = 0 to 16 {
      tmp = (u8)[in + i]; }
    in += 16;
    len -= 16; }

  for i = 0 to 16 {
    if i < len {
      tmp = (u8)[in + i]; }}

  return tmp;
}
```

Memory Calling Contract

$$\text{valid-mem}_{\text{load}}(\text{in}_0, \text{len}_0) = [\text{in}_0; \text{in}_0 + \text{len}_0]$$

Static Analysis

Variables in the Abstract Domain

Let $\mathcal{P}$ be a set of pointers. To a variable $x \in \mathcal{V}$, we associate:

- $x \in \mathcal{V}^\sharp$: its abstract value.
- $x_0 \in \mathcal{V}^\sharp$: its abstract initial value.
- $\mathrm{pt}_x \subseteq \mathcal{P}$: points-to information.
- $\mathrm{offset}_x \in \mathcal{V}^\sharp$: its abstract offset.

Static Analysis

Variables in the Abstract Domain

Let $\mathcal{P}$ be a set of pointers. To a variable $x \in \mathcal{V}$, we associate:

- $x \in \mathcal{V}^\sharp$: its abstract value.
- $x_0 \in \mathcal{V}^\sharp$: its abstract initial value.
- $\mathrm{pt}_x \subseteq \mathcal{P}$: points-to information.
- $\mathrm{offset}_x \in \mathcal{V}^\sharp$: its abstract offset.

Moreover, for every $\mathrm{p} \in \mathcal{P}$, we have:

- $\mathrm{mem}_\mathrm{p} \in \mathcal{V}^\sharp$: memory accesses at p (plus an offset).

Static Analysis

Concretization Function

We decompose x into a base pointer p and an offset offset_x:

$$\gamma(\mathsf{pt}_x = \{\mathsf{p}\} \wedge \mathsf{offset}_x = \mathcal{S}^\sharp) = x \mapsto \{\mathsf{p} + o \mid o \in \gamma(\mathcal{S}^\sharp)\}$$

Static Analysis

Concretization Function

We decompose x into a base pointer p and an offset offset_x:

$$\gamma(\mathsf{pt}_x = \{\mathsf{p}\} \wedge \mathsf{offset}_x = \mathcal{S}^\sharp) = x \mapsto \{\mathsf{p} + o \mid o \in \gamma(\mathcal{S}^\sharp)\}$$

Example

- $\gamma(\mathsf{pt}_x = \{\mathsf{p}\} \wedge \mathsf{offset}_x = [32; 63]) = x \mapsto [\mathsf{p} + 32; \mathsf{p} + 63]$

Static Analysis

Concretization Function

We decompose x into a base pointer p and an offset offset_x:

$$\gamma(\mathsf{pt}_x = \{\mathsf{p}\} \wedge \mathsf{offset}_x = \mathcal{S}^\sharp) = x \mapsto \{\mathsf{p} + o \mid o \in \gamma(\mathcal{S}^\sharp)\}$$

Example

- $\gamma(\mathsf{pt}_x = \{\mathsf{p}\} \wedge \mathsf{offset}_x = [32; 63]) = x \mapsto [\mathsf{p} + 32; \mathsf{p} + 63]$
- Abstract transformer:
  - $\mathcal{S}^\sharp$: $\mathsf{pt}_x = \{\mathsf{p}\} \wedge \mathsf{offset}_x = [32; 63]$
    $y \leftarrow x + 16$
  - $\mathcal{S}'^\sharp$: $\mathsf{pt}_y = \{\mathsf{p}\} \wedge \mathsf{offset}_y = [48; 79]$

Static Analysis

Remark

- In $y \leftarrow x + z$, we can either use x or z as a base pointer.
- In practice, it is never a problem (assembly coding style).

Static Analysis

Memory Calling Contract

Let f be a procedure with pointers $\mathcal{P}$. If:

$$[\![\mathsf{f}]\!]^\sharp(\mathcal{S}^\sharp_{\mathsf{init}}) \doteq \bigwedge_{\mathsf{p}\in\mathcal{P}} \mathsf{mem}_{\mathsf{p}} = \mathcal{S}^\sharp_{\mathsf{p}} \wedge \ldots$$

Then for every $\mathcal{S}_{\mathsf{init}} \subseteq \gamma(\mathcal{S}^\sharp_{\mathsf{init}})$:

$$\mathsf{valid\text{-}mem}_{\mathsf{f}}(\mathcal{S}_{\mathsf{init}}) \subseteq \bigcup_{\mathsf{p}\in\mathcal{P}} \gamma(\mathcal{S}^\sharp_{\mathsf{p}})$$

Static Analysis

Example

- $\mathcal{S}^\sharp \;:\; \mathrm{pt}_x = \{\mathrm{p}\} \wedge \mathrm{mem}_{\mathrm{p}} = [0;127] \wedge \mathrm{offset}_x = [128;128+16]$

  $\mathrm{tmp} \leftarrow (\mathrm{u8})[x+16]$

- $\mathcal{S}'^\sharp \;:\; \mathrm{mem}_{\mathrm{p}} = [0;127] \cup^\sharp [128;128+32] = [0;160]$

Example

```
fn load(reg u64 in, reg u64 len) {
  inline int i;
  reg u8 tmp;

  tmp = 0;
  while (len >= 16) {
    for i = 0 to 16 {
      tmp = (u8)[in + i]; }
    in += 16;
    len -= 16; }

  for i = 0 to 16 {
    if i < len {
      tmp = (u8)[in + i]; }}

  return tmp;
}
```

After the While Loop

$$\begin{aligned} & 0 \leq \mathsf{offset}_{\mathsf{in}}, \mathsf{len}, \mathsf{len}_0, \mathsf{mem}_{\mathsf{in}} \\ \wedge\ & \mathsf{offset}_{\mathsf{in}} + \mathsf{len} = \mathsf{len}_0 \\ \wedge\ & \mathsf{len}_0 - 15 \leq \mathsf{offset}_{\mathsf{in}} \leq \mathsf{len}_0 \\ \wedge\ & \mathsf{mem}_{\mathsf{in}} \leq \mathsf{offset}_{\mathsf{in}} \end{aligned}$$

At the End

$$0 \leq \mathsf{mem}_{\mathsf{in}} \leq \mathsf{len}_0$$

Static Analysis

The Analyzer

- Intervals + Relational domain (polyhedra).
- Basic syntactic pre-analysis.
- Disjunctive domain (using the control flow).
- Simple non-relational boolean abstractions (for bools and initialization).
- Brutal handling of function calls.

Static Analysis

Result

For Poly1305, with signature:

```
export fn poly1305_avx2(reg u64 out, reg u64 in, reg u64 len, reg u64 k)
```

We infer the ranges:

$$\begin{array}{llll} \mathrm{mem}_{\mathrm{out}}: & \mathrm{out} + [0;16[& \mathrm{mem}_{\mathrm{len}}: & \emptyset \\ \mathrm{mem}_{\mathrm{k}}\ : & \mathrm{k} + [0;32[& \mathrm{mem}_{\mathrm{in}}\ : & \mathrm{in} + [0;\mathrm{len}[\end{array}$$

Static Analysis

Caveat

We manually provide some information to the analyser:

- pointers (input) variables: k, in and out in Poly1305.
- relational (input) variables: len in Poly1305.

Conclusion

Conclusion

Contributions

A framework to build high-speed certified implementations of cryptographic primitives.

- Code is manually optimized.
- Functional correctness is obtained by game hopping.
- Safety and security against timing attacks are proved automatically.
- Efficient implementation of Poly1305, ChaCha20 and Gimli.

Conclusion

Future Works

- More TLS 1.3 primitives.
- More architectures, more general purpose language.
  - procedure calls.
  - register allocation/spilling.
- Certification for safety proofs.