



# Jasmin: high-assurance high-speed cryptography

---

Miguel Quaresma   Santiago Arranz Olmos

September 4, 2024

Max Planck Institute for Security and Privacy

# Efficient, correct, safe, and secure

```
fn memeq(reg u64 p q n) -> reg u64 {  
  reg u64 r one i;  
  r = 0;  
  one = 1;  
  i = 0;  
  while (i < n) {  
    if (r != 0) {  
      reg u64 a b;  
      a = [p];  
      b = [q];  
      r = a != b ? one : r;  
      p += 8;  
      q += 8;  
    }  
    i = #INC(i);  
  }  
  return r;  
}
```

# Efficient, correct, safe, and secure

```
fn memeq(reg u64 p q n) -> reg u64 {  
  reg u64 r one i;  
  r = 0;  
  one = 1;  
  i = 0;  
  while (i < n) {  
    if (r != 0) {  
      reg u64 a b;  
      a = [p];  
      b = [q];  
      r = a != b ? one : r;  
      p += 8;  
      q += 8;  
    }  
    i = #INC(i);  
  }  
  return r;  
}  
  
memeq:  
  movq $0, %rax  
  movq $1, %rcx  
  movq $0, %r8  
  jmp Lmemeq$1  
Lmemeq$2:  
  cmpq $0, %rax  
  je Lmemeq$3  
  movq (%rdi), %r9  
  movq (%rsi), %r10  
  cmpq %r10, %r9  
  cmovne %rcx, %rax  
  addq $8, %rdi  
  addq $8, %rsi  
Lmemeq$3:  
  incq %r8  
Lmemeq$1:  
  cmpq %rdx, %r8  
  jnb Lmemeq$2  
  ret
```

# Efficient, correct, safe, and secure

```
fn memeq(reg u64 p q n) -> reg u64 {  
  reg u64 r one i;  
  r = 0;  
  one = 1;  
  i = 0;  
  while (i < n) {  
    if (r != 0) {  
      reg u64 a b;  
      a = [p];  
      b = [q];  
      r = a != b ? one : r;  
      p += 8;  
      q += 8;  
    }  
    i = #INC(i);  
  }  
  return r;  
}  
  
memeq:  
  movq $0, %rax  
  movq $1, %rcx  
  movq $0, %r8  
  jmp Lmemeq$1  
Lmemeq$2:  
  cmpq $0, %rax  
  je Lmemeq$3  
  movq (%rdi), %r9  
  movq (%rsi), %r10  
  cmpq %r10, %r9  
  cmovne %rcx, %rax  
  addq $8, %rdi  
  addq $8, %rsi  
Lmemeq$3:  
  incq %r8  
Lmemeq$1:  
  cmpq %rdx, %r8  
  jnb Lmemeq$2  
  ret
```

# Efficient, correct, safe, and secure

```
fn memeq(reg u64 p q n) -> reg u64 {  
    reg u64 r one i;  
    r = 0;  
    one = 1;  
    i = 0;  
    while (i < n) {  
        if (r != 0) {  
            reg u64 a b;  
            a = [p];  
            b = [q];  
            r = a != b ? one : r;  
            p += 8;  
            q += 8;  
        }  
        i = #INC(i);  
    }  
    return r;  
}
```

```
memeq:  
    movq $0, %rax  
    movq $1, %rcx  
    movq $0, %r8  
    jmp Lmemeq$1  
Lmemeq$2:  
    cmpq $0, %rax  
    je Lmemeq$3  
    movq (%rdi), %r9  
    movq (%rsi), %r10  
    cmpq %r10, %r9  
    cmovne %rcx, %rax  
    addq $8, %rdi  
    addq $8, %rsi  
Lmemeq$3:  
    incq %r8  
Lmemeq$1:  
    cmpq %rdx, %r8  
    jb Lmemeq$2  
    ret
```

The diagram illustrates the mapping of Rust code to assembly code. Red arrows indicate control flow, and blue dashed arrows indicate data flow.

- The function signature `fn memeq(reg u64 p q n) -> reg u64 {` maps to the assembly label `memeq:` via a green dotted arrow.
- The variable declarations `reg u64 r one i;` map to the initial register setup in assembly: `movq $0, %rax` (for `r`), `movq $1, %rcx` (for `one`), and `movq $0, %r8` (for `i`).
- The `while (i < n) {` loop maps to the `jmp Lmemeq$1` instruction.
- The `if (r != 0) {` conditional maps to the `cmpq $0, %rax` and `je Lmemeq$3` instructions.
- The `reg u64 a b;` declaration maps to the `movq (%rdi), %r9` and `movq (%rsi), %r10` instructions.
- The `a = [p];` and `b = [q];` memory access operations map to the `addq $8, %rdi` and `addq $8, %rsi` instructions.
- The `r = a != b ? one : r;` conditional assignment maps to the `cmovne %rcx, %rax` instruction.
- The `p += 8;` and `q += 8;` pointer increments map to the `addq $8, %rdi` and `addq $8, %rsi` instructions.
- The `i = #INC(i);` increment operation maps to the `incq %r8` instruction.
- The `return r;` statement maps to the `cmpq %rdx, %r8` and `jb Lmemeq$2` instructions, which then lead to the `ret` instruction.



**Jasmin:** `github.com/jasmin-lang/jasmin`

**EasyCrypt specifications:** `github.com/formosa-crypto/crypto-specs`

**Libjade:** `github.com/formosa-crypto/libjade`