



Constant-Time Coding Support in LLVM: *Protecting Cryptographic Code at the Compiler Level*

LLVM 2025, 27th October, Julius Alexandre

The Compiler Optimization Problem

Modern compilers excel at making code run faster:

- Eliminate redundant operations
- Vectorize loops for parallel execution
- Restructure algorithms for performance

Break Cryptographic code...

Timing Attacks: The Silent Threat

Carefully crafted constant-time code:

Generated Assembly:

```
const bool cond = i == secret_idx;
const uint64_t mask = (- (int64_t)cond);
result |= table[i] & mask;
```



```
cmp    rdi, rcx    ; Compare i == secret_idx
je     .LBB0_3     ; BRANCH if equal (Timing leak!)
xor     edx, edx   ; else: mask = 0
jmp     .LBB0_4
```

```
if (i == secret_idx) then jmp
```

- [CVE-2022-4304](#) (OpenSSL RSA - billions affected)
- [CVE-2021-38153](#) (Apache Kafka authentication - Fortune 100 companies)
- [CVE-2023-5388](#) (NSS RSA ~150M Firefox users)
- And more...

Real-World Impact

ETH Zürich Study: "Breaking Bad" (2024)

- 8 production cryptographic libraries analyzed
- Tested on different compilers version for LLVM and GCC
- 44,604 experiments found compiler-induced vulnerabilities
- BearSSL, HACL*, Fiat-Crypto, BoringSSL, OpenSSL derivatives

Prior work:

- Simon and Chisnall `__builtin_ct_choose` (2018)
- Rust's optimization experiments

Cryptographers current solution:

- Using inline assembly
- Bitmask hack to bypass optimization
- Disable optimization

Our Solution: `__builtin_ct_select`

A new compiler intrinsic family:

```
result = __builtin_ct_select(condition,  
                             value_if_true,  
                             value_if_false);
```

Key Properties:

- ✓ Guarantees constant-time execution
- ✓ Preserved through all optimization levels
- ✓ Acts as optimization barrier
- ✓ Semantic meaning: "this must remain constant-time"
- ✓ All happening in the Post-RA

Circumvent Branch-base Timing Attacks

Carefully crafted constant-time code:

```
const bool cond = i == secret_idx;
result |= __builtin_ct_select(cond, table[i], 0);
```

x86 Assembly:

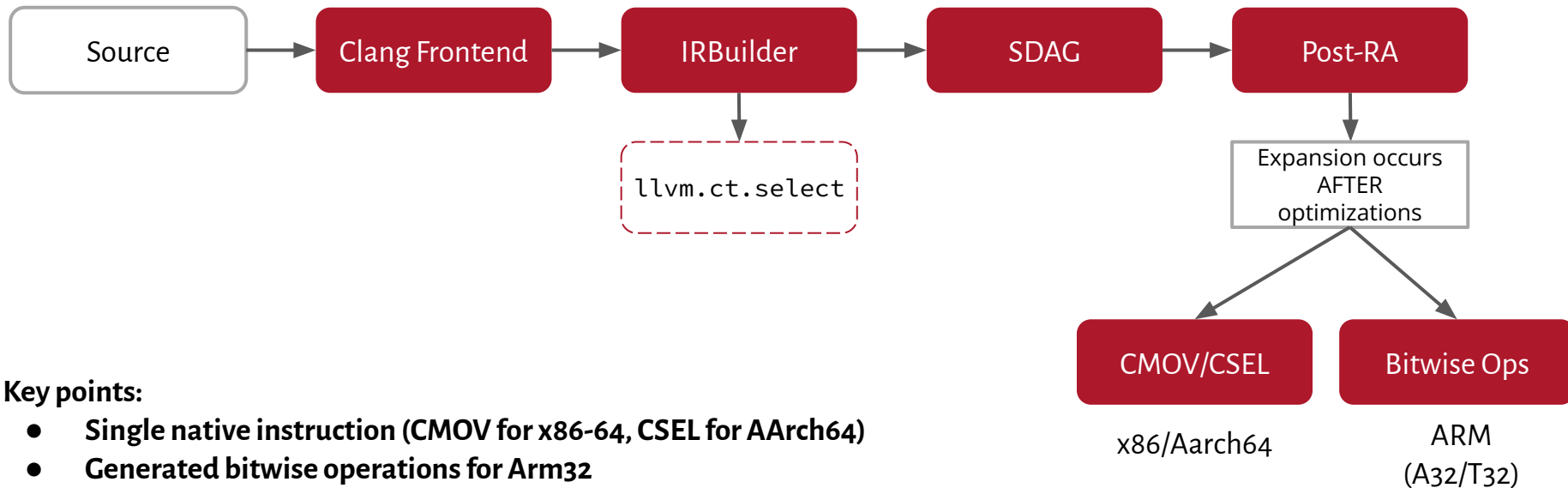
```
cmp    rdi, rcx          ; i == secret_idx
sete   dl                ; Set dl = 1
test   dl, dl            ; Test the condition
mov     edx, 0           ; Prepare edx = 0
cmovne rdx, [rsi + 8*rcx] ; CONDITIONAL MOVE
or      rax, rdx          ; result |= rdx
```

ARM32 Assembly:

```
sub     r4, r0, r2        ; Arithmetic comparison
rsbs    r5, r4, #0
adc     r10, r4, r5
rsb     r4, r10, #0       ; Create explicit mask
and     r5, r6, r4        ; Explicit masking
orr     r3, r5, r3        ; Unconditional OR
```

How It Works: Architecture Support

NATIVE SUPPORT (x86, Aarch64 & Arm32)

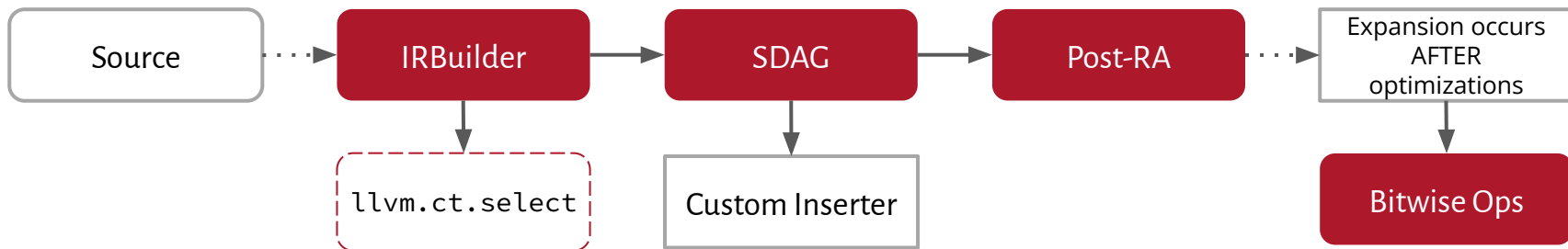


Key points:

- **Single native instruction (CMOV for x86-64, CSEL for AArch64)**
- **Generated bitwise operations for Arm32**
- **Constant-time enforced at Post-RA Expansion (AFTER all optimizations)**

How It Works: Architecture Support

NATIVE SUPPORT (i386)

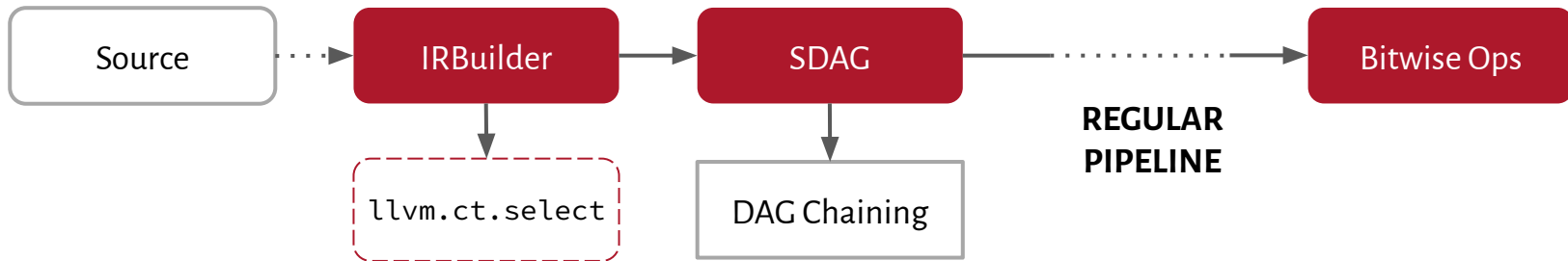


Key points:

- **Generates bitwise operations pattern (no CMOV available)**
- **UNIQUE two-phase approach (only architecture using both Custom Inserter + Post-RA)**

How It Works: Architecture Support

FALLBACK SUPPORT (RISC-V, Wasm, Mips, ...)



Key points:

- Generates bitwise operations for said Architecture
- Constant-time enforced at SelectionDAG level

From RFC to Implementation

Community Engagement:

- RFC published on LLVM Discourse (August 2025)
- Strong support from cryptography maintainers
- Valuable feedback from LLVM developers

Real-World benchmarking:

- Tested BoringSSL, OpenSSL, etc
- Worked across multiple Architectures
- Better results compared

Beyond C/C++: Language Support

LLVM-based languages can leverage this work:

Rust:

- Exploring intrinsics integration
- Safe wrappers in standard library

Swift:

- Apple can look into integrating our implementation

WebAssembly:

- Critical for browser-based cryptography

Challenges:

- GCC and Cranelift backend compatibility

What's Next?

Future Intrinsics:

- `__builtin_ct<op>` for arithmetic operations
- `__builtin_ct_expr` for entire expressions
- Memory operations and string comparisons

Goal: Make secure crypto practical in high-level languages

Key Takeaways



1. Compiler optimizations break constant-time guarantees
2. `__builtin_ct_select` provides compiler-level protection
3. Cross-architecture support
4. Community-driven approach with strong adoption
5. A crucial step toward practical secure cryptography