Fun with flags: How Compilers Break and Fix Constant-Time Code

Antoine Geimer

July 3st, 2025

Background: side-channels

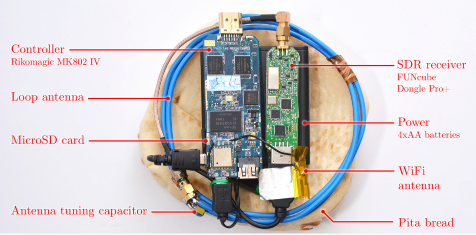
Definition

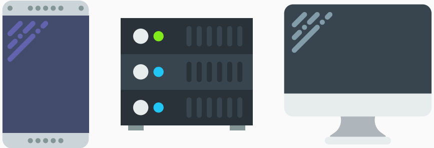
Side-channels are side-effects in a program's execution that can leak information

Background: side-channels

Definition

Side-channels are side-effects in a program's execution that can leak information



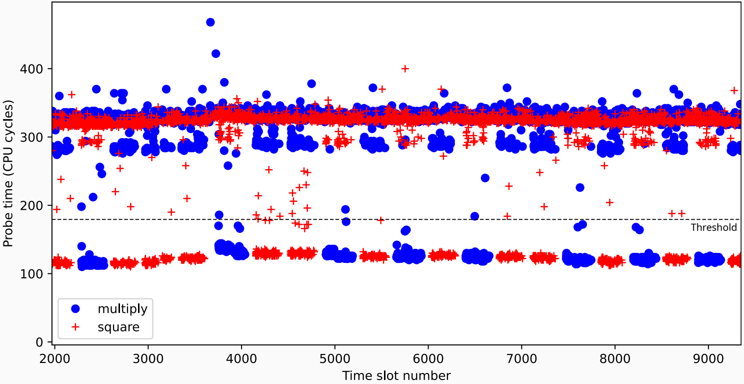


Hardware attacks
→ physical access

uarch attacks
→ co-located attacker

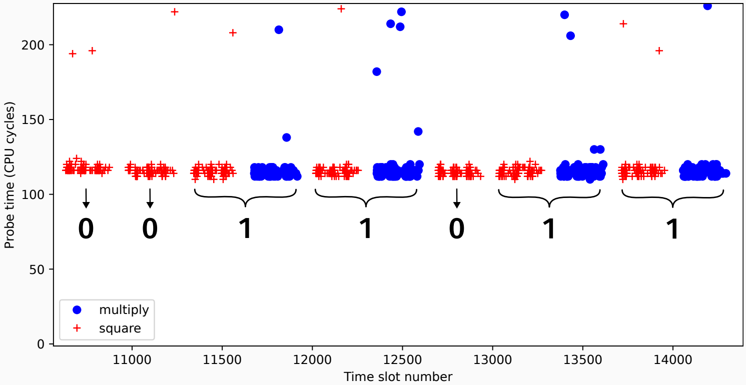
Example: RSA decryption

time cache accesses $\rightarrow$ get key $\rightarrow$ profit!



Example: RSA decryption

time cache accesses → get key → profit!



Constant-time programming

- Hardware cause: components shared between processes
  $\rightarrow$ unlikely to be fixed

Constant-time programming

- Hardware cause: components shared between processes
  $\rightarrow$ unlikely to be fixed
- Software countermeasure: constant-time programming
- Ensuring the microarchitectural state is independent of secret values

Constant-time programming

- Hardware cause: components shared between processes
  $\rightarrow$ unlikely to be fixed
- Software countermeasure: constant-time programming
- Ensuring the microarchitectural state is independent of secret values

Basically: no secret-dependent branch or memory accesses

Constant-time in practice[1]

Example in Kyber:

```
void poly_frommsg(int16_t r[SIZE], uint8_t msg[32]) {
  int16_t mask;

  for (int i = 0; i<SIZE/8; i++) {
    for (int j = 0; j<8; j++) {
      if ((msg[i] >> j) & 1)
        r[8*i+j] = CONSTANT;
      else
        r[8*i+j] = 0;
    }
  }
}
```

[1] From Antoon Purnal: https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/hqbtIGFKIpU

Constant-time in practice[1]

Example in Kyber:

```
void poly_frommsg(int16_t r[SIZE], uint8_t msg[32]) {
  int16_t mask;

  for (int i = 0; i<SIZE/8; i++) {
    for (int j = 0; j<8; j++) {
      if ((msg[i] >> j) & 1)
        r[8*i+j] = CONSTANT;
      else
        r[8*i+j] = 0;
    }
  }
}
```

not CT!

[1] From Antoon Purnal: https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/hqbtIGFKIpU

Constant-time in practice[2]

Example in Kyber:

```c
void poly_frommsg(int16_t r[SIZE], uint8_t msg[32]) {
  int16_t mask;

  for (int i = 0; i<SIZE/8; i++) {
    for (int j = 0; j<8; j++) {
      mask = -(int16_t)((msg[i] >> j) & 1); // bitmask arithmetic
      r[8*i+j] = mask & CONSTANT;
    }
  }
}
```

C source: **CT**

[2] From Antoon Purnal: `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/hqbtIGFKIpU`

Constant-time vs compilers: example (2)

Compiled with LLVM:

```
    xor     eax, eax
.outer:
    xor     ecx, ecx
.inner:
    movzx   r8d, ptr [rsi+rax]
    xor     edx, edx
    bt      r8d, ecx
    jae     .skip
    mov     edx, CONSTANT
.skip:
    ; [...]
    jne     .inner
    ; [...]
    jne     .outer
```

Constant-time vs compilers: example (2)

Compiled with LLVM:

```
    xor     eax, eax
.outer:
    xor     ecx, ecx
.inner:
    movzx   r8d, ptr [rsi+rax]
    xor     edx, edx
    bt      r8d, ecx
    jae     .skip
    mov     edx, CONSTANT
.skip:
    ; [...]
    jne     .inner
    ; [...]
    jne     .outer
```

→ secret-dependent branch

Constant-time vs compilers: example (2)

Compiled with LLVM:

```
    xor     eax, eax
.outer:
    xor     ecx, ecx
.inner:
    movzx   r8d, ptr [rsi+rax]
    xor     edx, edx
    bt      r8d, ecx
    jae     .skip
    mov     edx, CONSTANT
.skip:
    ; [...]
    jne     .inner
    ; [...]
    jne     .outer
```

→ secret-dependent branch

Compiled with GCC:

```
    mov     edx, 0
.outer:
    mov     ecx, 0
.inner:
    movzx   eax, ptr [rsi]
    sar     eax, cl
    and     eax, 1
    neg     eax
    and     ax, CONSTANT
    mov     ptr [rdi+rcx*2], ax
    ; [...]
    jne     .inner
    ; [...]
    jne     .outer
```

Constant-time vs compilers: example (2)

Compiled with LLVM:

```
    xor     eax, eax
.outer:
    xor     ecx, ecx
.inner:
    movzx   r8d, ptr [rsi+rax]
    xor     edx, edx
    bt      r8d, ecx
    jae     .skip
    mov     edx, CONSTANT
.skip:
    ; [...]
    jne     .inner
    ; [...]
    jne     .outer
```

→ secret-dependent branch

Compiled with GCC:

```
    mov     edx, 0
.outer:
    mov     ecx, 0
.inner:
    movzx   eax, ptr [rsi]
    sar     eax, cl
    and     eax, 1
    neg     eax
    and     ax, CONSTANT
    mov     ptr [rdi+rcx*2], ax
    ; [...]
    jne     .inner
    ; [...]
    jne     .outer
```

→ still CT

Related work

Known problem... but few studies:

- either limited to short snippets or older i386 programs[3]
- or providing only quantitative insights[4][5]

[3] Laurent Simon et al. "What You Get Is What You C: Controlling Side Effects in Mainstream C Compilers". In: *EuroS&P*. 2018.
[4] Moritz Schneider et al. *Breaking Bad: How Compilers Break Constant-Time~Implementations*. 2024.
[5] Lukas Gerlach et al. "Do Compilers Break Constant-time Guarantees?" In: *FC*. 2025.

Related work

Known problem... but few studies:

- either limited to short snippets or older i386 programs[3]
- or providing only quantitative insights[4][5]

→ lacking qualitative studies

How do compilers break CT guarantees?

[3] Laurent Simon et al. “What You Get Is What You C: Controlling Side Effects in Mainstream C Compilers”. In: *EuroS&P*. 2018.
[4] Moritz Schneider et al. *Breaking Bad: How Compilers Break Constant-Time~Implementations*. 2024.
[5] Lukas Gerlach et al. “Do Compilers Break Constant-time Guarantees?” In: *FC*. 2025.

Research questions

RQs:

- **RQ1** How can we detect compiler-introduced CT leakages?
- **RQ2** Which compiler optimizations introduce them?
- **RQ3** Can we prevent such leakages while preserving performance?

Research questions

RQs:

- **RQ1** How can we detect compiler-introduced CT leakages?
- **RQ2** Which compiler optimizations introduce them?
- **RQ3** Can we prevent such leakages while preserving performance?

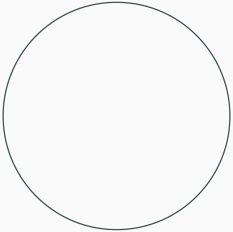
Contributions

- Simple methodology to detect such bugs using Microwalk
- Analysis of how optimization passes interact to break CT
- Evaluation of a simple defense: disabling such optimizations

Challenge: lack of ground truth

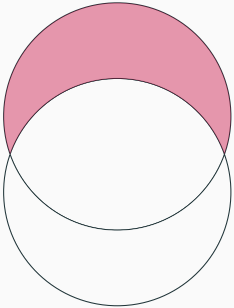
A two-fold problem:

binary CT violations



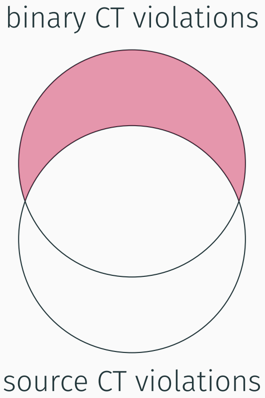
Challenge: lack of ground truth

A two-fold problem:



Challenge: lack of ground truth

A two-fold problem:



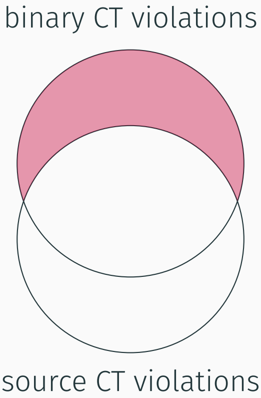
Potential solution: only analyze verified libraries

→ risks limiting experiment's scope

→ developers often use non-verified libraries

Challenge: lack of ground truth

A two-fold problem:



Potential solution: only analyze verified libraries

→ risks limiting experiment's scope

→ developers often use non-verified libraries

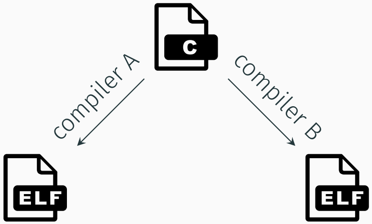
…or apply manual filtering?

→ done in Schneider et al.

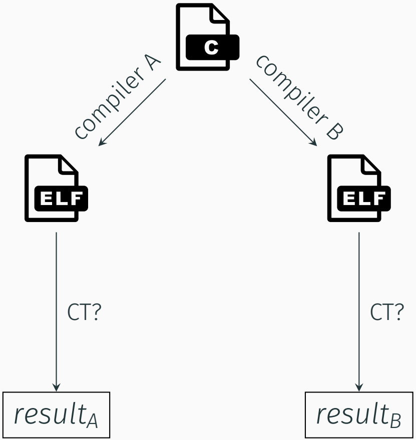
→ risks missing leakages

→ thwarted by function inlining

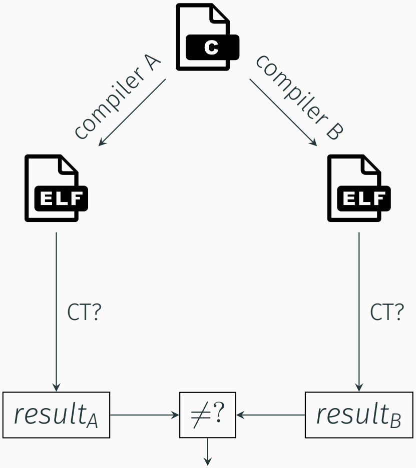
Our approach: differential testing



Our approach: differential testing

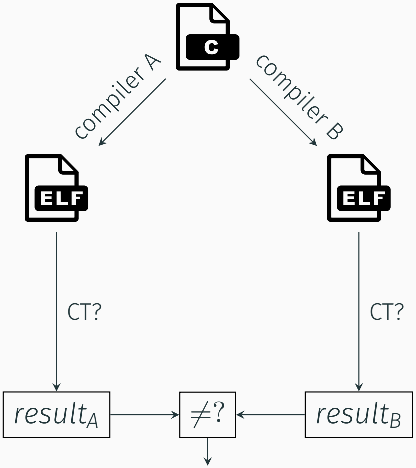


Our approach: differential testing



compiler-introduced CT violations

Our approach: differential testing



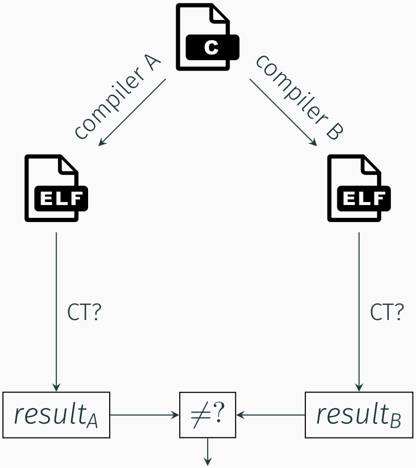
compiler-introduced CT violations

Choosing the right metric for comparison:

→ Schneider et al. : % of vuln. binaries

→ number vulnerable instructions: impacted by inlining and loop unrolling

Our approach: differential testing



compiler-introduced CT violations

Choosing the right metric for comparison:

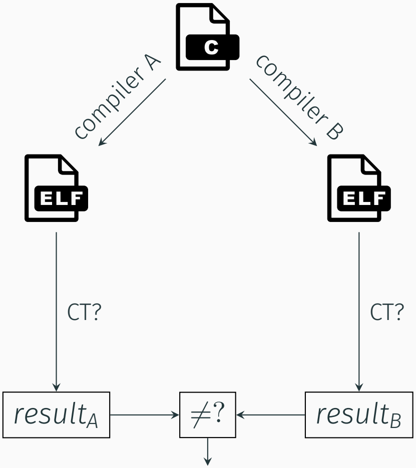
→ Schneider et al. : % of vuln. binaries

→ number vulnerable instructions: impacted by inlining and loop unrolling

→ solution: compare source code lines

→ we use DWARF debugging symbols

Implementation



compiler-introduced CT violations

Source benchmarks

MbedTLS and BearSSL from previous works

Compilers

LLVM 12/18 and GCC 9/13, O3 and Os

CT detection

Dynamic approach: Microwalk

Results

Binaries	LLVM O3 v12	LLVM O3 v18	GCC O3 v9	GCC O3 v13
RSA-mbedtls (PKCS)	47	47	52	**48** ▼
RSA-mbedtls (OAEP)	46	**48** ▲	49	49
ECDSA-mbedtls	60	**64** ▲	61	**62** ▲
RSA-bearssl (OAEP)	0	**1** ▲	0	0
ECDSA-bearssl	0	**1** ▲	0	0
poly_frommsg	0	**1** ▲	0	0
jump_threading	0	0	1	1
loop_unswitching	1	1	1	1
path_splitting	0	0	1	1

Results

Binaries	LLVM O3 v12	LLVM O3 v18	GCC O3 v9	GCC O3 v13
RSA-mbedtls (PKCS)	47	47	52	**48** ▼
RSA-mbedtls (OAEP)	46	**48** ▲	49	49
ECDSA-mbedtls	60	**64** ▲	61	**62** ▲
RSA-bearssl (OAEP)	0	**1** ▲	0	0
ECDSA-bearssl	0	**1** ▲	0	0
poly_frommsg	0	**1** ▲	0	0
jump_threading	0	0	1	1
loop_unswitching	1	1	1	1
path_splitting	0	0	1	1

→ LLVM: general increase in newer versions

Results

Binaries	LLVM O3 v12	LLVM O3 v18	GCC O3 v9	GCC O3 v13
RSA-mbedtls (PKCS)	47	47	52	**48** ▼
RSA-mbedtls (OAEP)	46	**48** ▲	49	49
ECDSA-mbedtls	60	**64** ▲	61	**62** ▲
RSA-bearssl (OAEP)	0	**1** ▲	0	0
ECDSA-bearssl	0	**1** ▲	0	0
poly_frommsg	0	**1** ▲	0	0
jump_threading	0	0	1	1
loop_unswitching	1	1	1	1
path_splitting	0	0	1	1

→ LLVM: general increase in newer versions

→ not so much for GCC

→ both compilers can break CT

Pass analysis (1)

We analyzed the detected CT violations using **Compiler Explorer**:

→ OptPipeline tool allows us to isolate problematic passes

→ GCC and LLVM break CT in different ways: code patterns and optimizations

→ Limitation: manual analysis

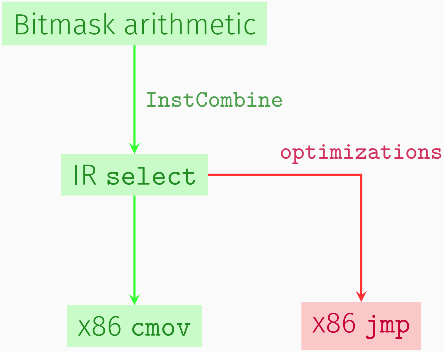
Pass analysis (2)

Different pathways to breaking CT...

Pass analysis (2)

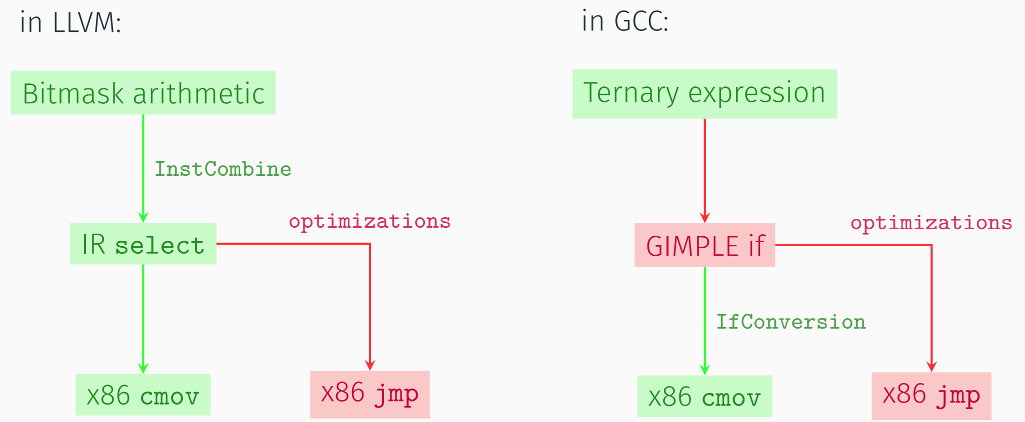
Different pathways to breaking CT...

in LLVM:



Pass analysis (2)

Different pathways to breaking CT...



Example: RSA-bearssl in LLVM (1)

Goal: perform a CT array access for windowed RSA modular exponentiation

```
for (int u = 1; u < N; u++) {
  uint32_t m;

  m = -EQ(u, secret);
  for (int v = 1; v < M; v++) {
    t2[v] |= m & base[v];
  }
  base += M;
}
```

C source

Example: RSA-bearssl in LLVM (1)

Goal: perform a CT array access for windowed RSA modular exponentiation

```c
for (int u = 1; u < N; u++) {
  uint32_t m;

  m = -EQ(u, secret);
  for (int v = 1; v < M; v++) {
    t2[v] |= m & base[v];
  }
  base += M;
}
```

C source

Inlining

InstCombine

```c
for (int u = 1; u < N; u++) {
  uint32_t m;

  m = (u == secret);
  for (int v = 1; v < M; v++) {
    t2[v] |= select(m, base[v], 0);
  }
  base += M;
}
```

LLVM IR (represented as C for clarity)

Example: RSA-bearssl in LLVM (2)

This transformation *by itself* is safe...

```
for (int u = 1; u < N; u++) {
  uint32_t m;

  m = (u == secret);
  for (int v = 1; v < M; v++) {
    t2[v] |= select(m, base[v], 0);
  }
  base += M;
}
```

Example: RSA-bearssl in LLVM (2)

This transformation *by itself* is safe... but allows further unsafe optimizations!

```c
for (int u = 1; u < N; u++) {
  uint32_t m;

  m = (u == secret);
  for (int v = 1; v < M; v++) {
    t2[v] |= select(m, base[v], 0);
  }
  base += M;
}
```

LoopUnswitch

```c
for (int u = 1; u < k; u++) {
  uint32_t m;

  m = (u == secret);
  if (m) {
    for (int v = 1; v < M; v++) {
      t2[v] |= base[v];
    }
  }
  base += M;
}
```

Example: RSA-bearssl in LLVM (2)

This transformation *by itself* is safe... but allows further unsafe optimizations!

```c
for (int u = 1; u < N; u++) {
  uint32_t m;

  m = (u == secret);
  for (int v = 1; v < M; v++) {
    t2[v] |= select(m, base[v], 0);
  }
  base += M;
}
```

CmovConversion

```c
for (int u = 1; u < k; u++) {
  uint32_t m;

  m = (u == secret);
  for (int v = 1; v < M; v++) {
    if (m) {
      t2[v] |= base[v];
    }
  }
  base += M;
}
```

Mitigations

We investigate a simple mitigation: disabling problematic optimizations

→ using (sometimes undocumented) compiler flags

→ GCC: we disable loop unswitching, jump threading and path splitting

→ LLVM: we disable loop unswitching, loop vectorization and cmov conversion

Mitigations

We investigate a simple mitigation: disabling problematic optimizations

→ using (sometimes undocumented) compiler flags

→ GCC: we disable loop unswitching, jump threading and path splitting

→ LLVM: we disable loop unswitching, loop vectorization and cmov conversion

Evaluation

→ effectiveness: rerun our benchmarks compiled with the mitigating flags

→ performance: reusing the libraries' existing performance benchmarks

Results

Binaries \ Mitig.?	LLVM O3 No	LLVM O3 Yes	GCC O3 No	GCC O3 Yes
RSA-mbedtls (PKCS)	47	**46** ▼	48	**50** ▲
RSA-mbedtls (OAEP)	48	**46** ▼	49	49
ECDSA-mbedtls	64	**61** ▼	62	62
RSA-bearssl (OAEP)	1	**0** ▼	0	0
ECDSA-bearssl	1	**0** ▼	0	0
poly_frommsg	1	**0** ▼	0	0
jump_threading	0	0	1	**0** ▼
loop_unswitching	1	**0** ▼	1	**0** ▼
path_splitting	0	0	1	**0** ▼

Results

Binaries \ Mitig.?	LLVM O3		GCC O3	
	No	Yes	No	Yes
RSA-mbedtls (PKCS)	47	**46** ▼	48	**50** ▲
RSA-mbedtls (OAEP)	48	**46** ▼	49	49
ECDSA-mbedtls	64	**61** ▼	62	62
RSA-bearssl (OAEP)	1	**0** ▼	0	0
ECDSA-bearssl	1	**0** ▼	0	0
poly_frommsg	1	**0** ▼	0	0
jump_threading	0	0	1	**0** ▼
loop_unswitching	1	**0** ▼	1	**0** ▼
path_splitting	0	0	1	**0** ▼

- Decrease in vulnerability
- CT binaries remain CT

Results

Binaries \ Mitig.?	LLVM O3 No	LLVM O3 Yes	GCC O3 No	GCC O3 Yes
RSA-mbedtls (PKCS)	47	**46** ▼	48	**50** ▲
RSA-mbedtls (OAEP)	48	**46** ▼	49	49
ECDSA-mbedtls	64	**61** ▼	62	62
RSA-bearssl (OAEP)	1	**0** ▼	0	0
ECDSA-bearssl	1	**0** ▼	0	0
poly_frommsg	1	**0** ▼	0	0
jump_threading	0	0	1	**0** ▼
loop_unswitching	1	**0** ▼	1	**0** ▼
path_splitting	0	0	1	**0** ▼

- Decrease in vulnerability
- CT binaries remain CT
- Negligible performance impact
  - → BearSSL: -3.30% (GCC), -0.43% (LLVM)
  - → MbedTLS: -0.71% (GCC), -1.14% (LLVM)

Future work

Our work has some limitations:

- benchmarks restricted to a few primitives
- optimization pipeline analysis is still manual

[6] Zhiyuan Zhang and Gilles Barthe. *CT-LLVM: Automatic Large-Scale Constant-Time Analysis*. 2025. URL: `https://eprint.iacr.org/2025/338`.

Future work

Our work has some limitations:

- benchmarks restricted to a few primitives
- optimization pipeline analysis is still manual

→ our list of problematic passes is incomplete

[6] Zhiyuan Zhang and Gilles Barthe. *CT-LLVM: Automatic Large-Scale Constant-Time Analysis*. 2025. URL: `https://eprint.iacr.org/2025/338`.

Future work

Our work has some limitations:

- benchmarks restricted to a few primitives
- optimization pipeline analysis is still manual

$\rightarrow$ our list of problematic passes is incomplete

Possible solution: applying an IR-level detection tool between each pass

- CT-LLVM[6]: not yet open-source

[6] Zhiyuan Zhang and Gilles Barthe. *CT-LLVM: Automatic Large-Scale Constant-Time Analysis*. 2025. URL: `https://eprint.iacr.org/2025/338`.

Future work

Our work has some limitations:

- benchmarks restricted to a few primitives
- optimization pipeline analysis is still manual

→ our list of problematic passes is incomplete

Possible solution: applying an IR-level detection tool between each pass

- CT-LLVM[6]: not yet open-source
- RQ: how do we generalize this to various LLVM backends?
- RQ: what about GCC?

[6] Zhiyuan Zhang and Gilles Barthe. *CT-LLVM: Automatic Large-Scale Constant-Time Analysis*. 2025. URL: `https://eprint.iacr.org/2025/338`.

Conclusion

We conducted a qualitative study of compiler-introduced CT violations:

- we introduced a simple detection methodolody based on differential testing
- we found multiple optimizations susceptible to break CT
- we suggest a simple and readily-deployable mitigation: just disabling them!
- we show this approach prevent the leakage we detected, with minimal overhead

Conclusion

We conducted a qualitative study of compiler-introduced CT violations:

- we introduced a simple detection methodolody based on differential testing
- we found multiple optimizations susceptible to break CT
- we suggest a simple and readily-deployable mitigation: just disabling them!
- we show this approach prevent the leakage we detected, with minimal overhead

Artifact repo: `https://github.com/ageimer/fun-with-flags`