UNIVERSITÄT PADERBORN

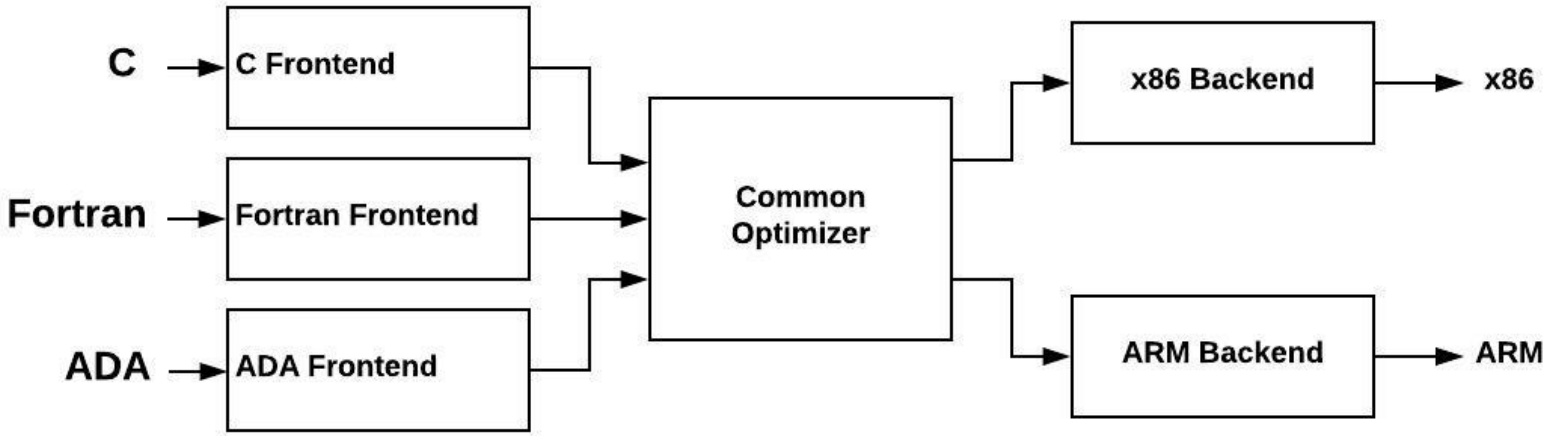
Heinz Nixdorf Institute

Security Implications Of Compiler Optimizations On Cryptography — A Review

UNIVERSITÄT PADERBORN

Compiler

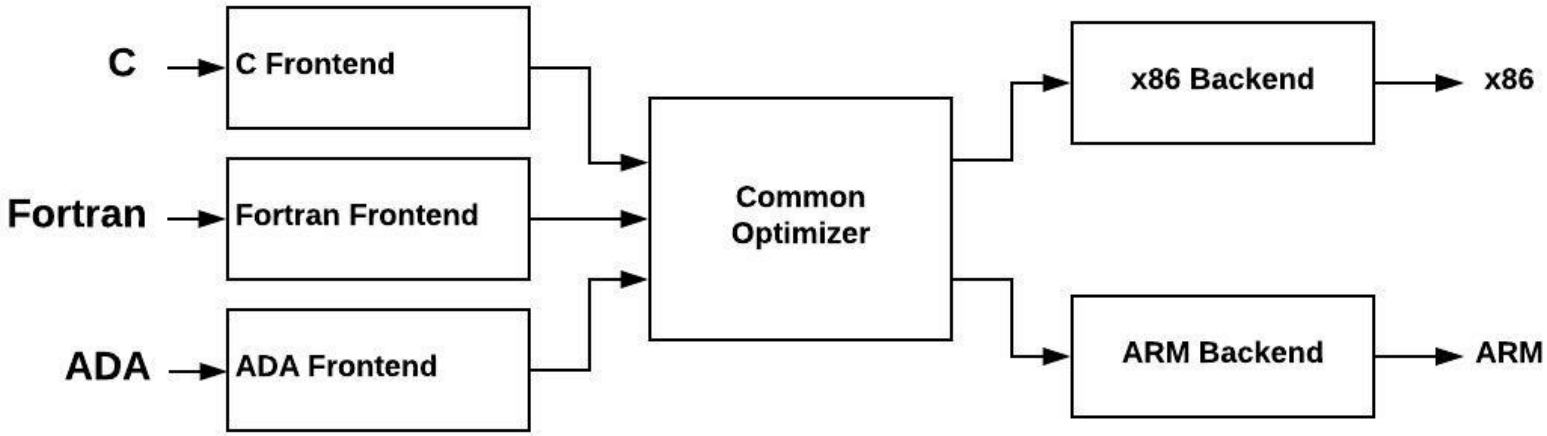
- Translates high-level abstract instructions to machine level



UNIVERSITÄT PADERBORN

Compiler

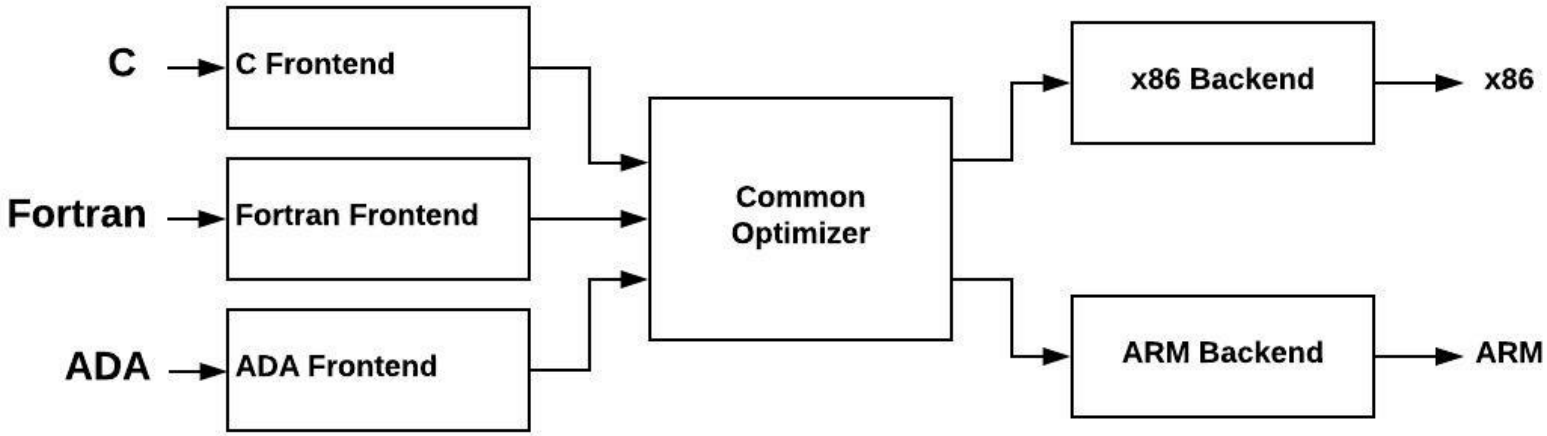
- Translates high-level abstract instructions to machine level
- Three Phase Compiler Architecture



UNIVERSITÄT PADERBORN

Compiler

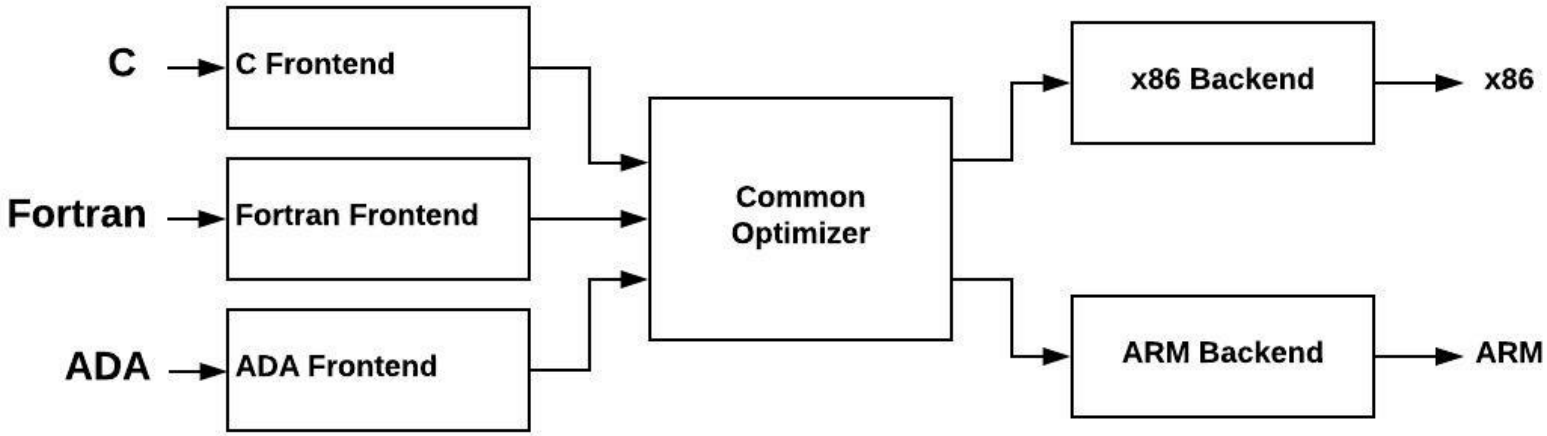
- Translates high-level abstract instructions to machine level
- Three Phase Compiler Architecture
- Frontend – Optimizer – Backend



UNIVERSITÄT PADERBORN

Compiler

- Translates high-level abstract instructions to machine level
- Three Phase Compiler Architecture
- Frontend – Optimizer – Backend
- **Here:** Clang – LLVM – X86



UNIVERSITÄT PADERBORN

Compiler optimization

Optimization Levels

- **Higher levels:** longer compile time, but faster run time
- Compiler flag: `-O1`, `-O2`, `-O3`

UNIVERSITÄT PADERBORN

Compiler optimization

Optimization Levels

- **Higher levels:** longer compile time, but faster run time
- Compiler flag: `-O1, -O2, -O3`

Dead Store Elimination

- Removes unused or overwritten memory store operations
- Compiler flag: `-fdse`

UNIVERSITÄT PADERBORN

Compiler optimization

Optimization Levels

Dead Store

```
1 int foo(int x)
2 {
3   int b = 2; // Dead Variable
4   int c = 100;
5   return c*x;
6 }
```

UNIVERSITÄT PADERBORN

Compiler optimization

Optimization Levels

- **Higher levels:** longer compile time, but faster run time
- Compiler flag: `-O1, -O2, -O3`

Dead Store Elimination

- Removes unused or overwritten memory store operations
- Compiler flag: `-fdse`

Function Inlining

- Eliminate function call overhead
- Compiler flag: `-finline-functions`

UNIVERSITÄT PADERBORN

Compiler optimization

Function Inlining

```
1  void f(int *x)
2  {
3    *x = x + 10;
4  }
5
6  void caller()
7  {
8    int a = 1;
9    f(&a); //Funtion Call
10   //   *&a = a + 10;  <— Can be inlined
11 }
```

- Eliminate function call overhead
- Compiler flag: `-finline-functions`

Problem...?

Problem...?
Implicit Requirements

UNIVERSITÄT PADERBORN

Implicit Requirements

Constant-Time Selection

- **Requirement:** Select between two branches based on a boolean value, **in constant time**

UNIVERSITÄT PADERBORN

Implicit Requirements

Constant-Time Selection

- **Requirement:** Select between two branches based on a boolean value, **in constant time**
- Generated code might contain jump instruction

UNIVERSITÄT PADERBORN

Implicit Requirements

Constant-Time Selection

- **Requirement:** Select between two branches based on a boolean value, **in constant time**
- Generated code might contain jump instruction
- **Timing attacks**: Branch prediction and Pipeline stalling

UNIVERSITÄT PADERBORN

Implicit Requirements

Constant-Time Selection

```
1 int conditional_select(bool b, int x, int y){
2   if(b){
3     return x;
4   }
5   else {
6     return y;
7   }
8 }
```

Listing 1: C Code

UNIVERSITÄT PADERBORN

Implicit Requirements

Constant-Time Selection

```
1  ; clang 3.9 -O3  -m32 -march=i386
2  conditional_select(bool, int, int):
3    mov     al, byte ptr [esp + 4]
4    test    al, al
5    jne     .LBB0_1   ; <---- JUMP
6    lea     eax, [esp + 12]
7    mov     eax, dword ptr [eax]
8    ret
9
10 .LBB0_1:
11   lea     eax, [esp + 8]
12   mov     eax, dword ptr [eax]
13   ret
```

Listing 2: Assembly Code

UNIVERSITÄT PADERBORN

Implicit Requirements

Secret Erasure

- **Requirement:** Erasing sensitive keys from memory after use

UNIVERSITÄT PADERBORN

Implicit Requirements

Secret Erasure

- **Requirement:** Erasing sensitive keys from memory after use
- Common technique to reset memory: `memset`

UNIVERSITÄT PADERBORN

Implicit Requirements

Secret Erasure

- **Requirement:** Erasing sensitive keys from memory after use
- Common technique to reset memory: `memset`
- Dead store elimination will optimize **useless*** calls to `memset`

UNIVERSITÄT PADERBORN

Implicit Requirements

Secret Erasure

```
1  int dummy(int x){
2    int y = x+1;
3    return y;
4  }
5
6  int secret_function(){
7    int key = 0xDEADBEEF;
8    int y = dummy(key);
9    key = 0x00;
10   return y;
11 }
```

Listing 3: C Code

UNIVERSITÄT PADERBORN

Implicit Requirements

Secret Erasure

```
1  ; clang 3.9 -O1  -m32 -march=i386
2
3  dummy(int):
4    mov      eax, dword ptr [esp + 4]
5    inc      eax
6    ret
7
8  secret_function():
9    sub      esp, 12
10   mov      dword ptr [esp], -559038737
11   call     dummy(int)
12   add      esp, 12 ; <---- Missing mov 0 (Optimized)
13   ret
```

Listing 4: Assembly Code

But,
What are the developers doing?

UNIVERSITÄT PADERBORN

Controlling the side-effects

UNIVERSITÄT PADERBORN

Controlling the side-effects

1. Custom Functions For Constant-Time Selection

UNIVERSITÄT PADERBORN

Controlling the side-effects

1. Custom Functions For Constant-Time Selection
2. Custom Functions For Stack Erasure

UNIVERSITÄT PADERBORN

Controlling the side-effects

1. Custom Functions For Constant-Time Selection
2. Custom Functions For Stack Erasure
3. Disabling Optimization

UNIVERSITÄT PADERBORN

Case Studies

[Bearssl, Monocypher, Libsodium, Crypto++, Libgcrypt ..]

UNIVERSITÄT PADERBORN

Case Studies

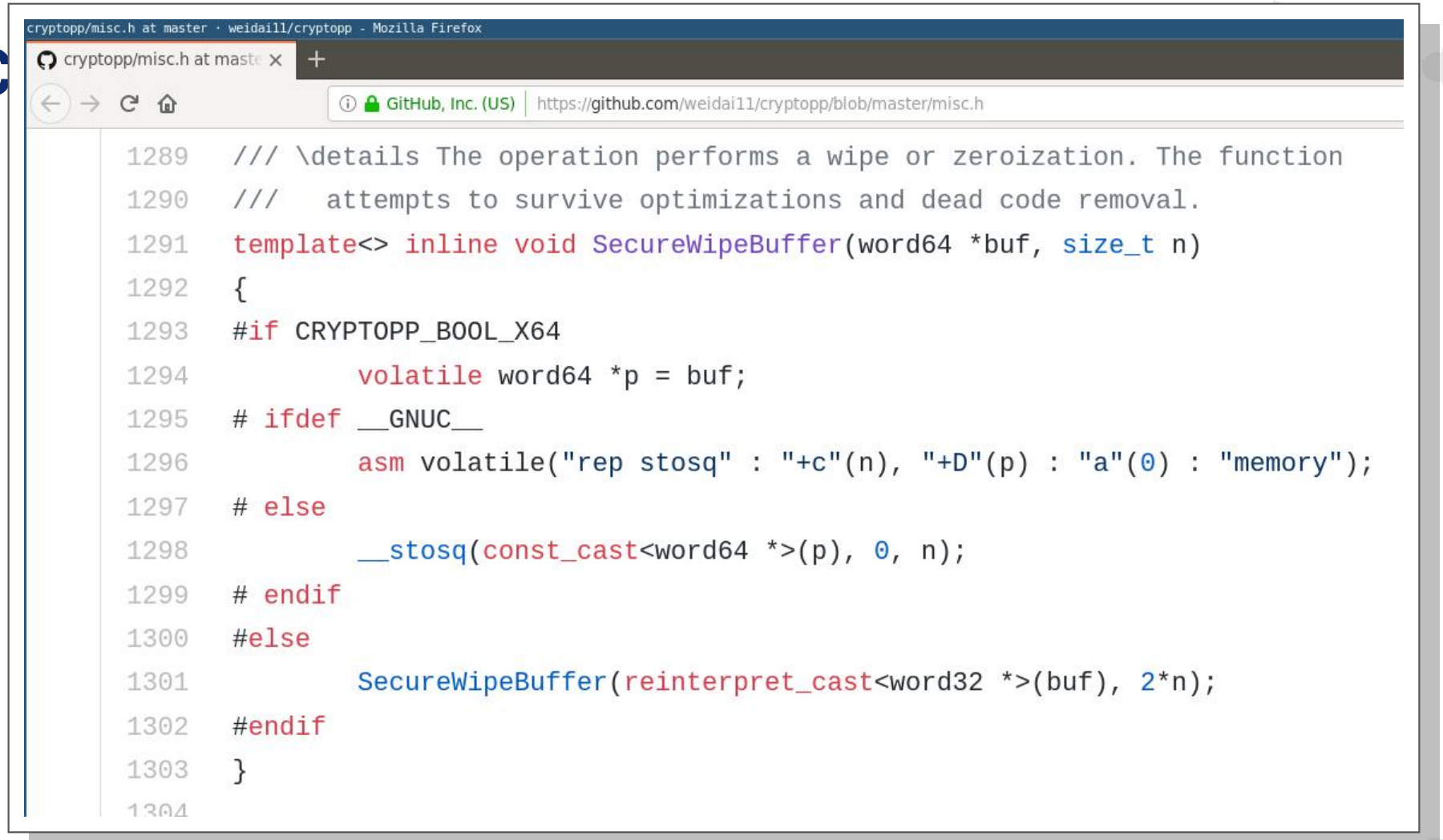
[Bearssl, Monocypher, Libsodium, Crypto++, Libgcrypt ..]

Secure memory erasure

- Different techniques to reach common goal
- **OpenSSL:** Inline assembly to avoid optimization
- Some projects use Platform provided functions

UNIVERSITÄT PADERBORN

C




```
1289  /// \details The operation performs a wipe or zeroization. The function
1290  ///   attempts to survive optimizations and dead code removal.
1291  template<> inline void SecureWipeBuffer(word64 *buf, size_t n)
1292  {
1293  #if CRYPTOPP_BOOL_X64
1294          volatile word64 *p = buf;
1295  # ifdef __GNUC__
1296          asm volatile("rep stosq" : "+c"(n), "+D"(p) : "a"(0) : "memory");
1297  # else
1298          __stosq(const_cast<word64 *>(p), 0, n);
1299  # endif
1300  #else
1301          SecureWipeBuffer(reinterpret_cast<word32 *>(buf), 2*n);
1302  #endif
1303  }
1304
```

UNIVERSITÄT PADERBORN

Case Studies

[Bearssl, Monocypher, Libsodium, Crypto++, Libgcrypt ..]

Secure memory erasure

- Different techniques to reach common goal
- **OpenSSL:** Inline assembly to avoid optimization
- Some projects use Platform provided functions

Constant Time Selection

- Same situation
- Custom functions to implement constant time selection

What now?

What now?
Add Compiler Support

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Implementation - Constant-Time Selection

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Implementation - Constant-Time Selection

```
__builtin_ct_choose(bool condition, Type x, Type y)
```

- Authors implement a built-in function in the Clang/LLVM framework

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Implementation - Constant-Time Selection

```
__builtin_ct_choose(bool condition, Type x, Type y)
```

- Authors implement a built-in function in the Clang/LLVM framework
- **x86_64 backend:** Compiled into a `CMOV` operation

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Implementation - Constant-Time Selection

```
__builtin_ct_choose(bool condition, Type x, Type y)
```

- Authors implement a built-in function in the Clang/LLVM framework
- **x86_64 backend:** Compiled into a `CMOV` operation
- For other backends: Compiled to `XOR`

[Simon et al. 2018]

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Evaluation - Constant-Time Selection

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Evaluation - Constant-Time Selection

- Authors verified the OpenSSL and mbedTLS

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Evaluation - Constant-Time Selection

- Authors verified the OpenSSL and mbedTLS
- Empirical verification - “**Dudect**” tool

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Evaluation - Constant-Time Selection

- Authors verified the OpenSSL and mbedTLS
- Empirical verification - “**Dudect**” tool
- **Overhead:**
  - Two implementations compared (100x)

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Evaluation - Constant-Time Selection

- Authors verified the OpenSSL and mbedTLS
- Empirical verification - “**Dudect**” tool
- **Overhead:**
  - Two implementations compared (100x)
  - OpenSSL’s elliptic curve - **1% overhead**
  - Custom RSA implementation - **4% Faster**

[Simon et al. 2018]

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Secret Erasure

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Secret Erasure

1. **Function-Based Solution**
   Performs stack erasure for every sensitive function and its callees.

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Secret Erasure

1. **Function-Based Solution**
   Performs stack erasure for every sensitive function and its callees.
2. **Stack-Based Solution**
   Callees only keep track of stack usage and the sensitive function does the erasure, only once.

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Secret Erasure

1. **Function-Based Solution**
   Performs stack erasure for every sensitive function and its callees.
2. **Stack-Based Solution**
   Callees only keep track of stack usage and the sensitive function does the erasure, only once.
3. **Call-Graph Based Solution**
   The call graph is used to determine the maximum stack usage of a sensitive function at compilation time.

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Secret Erasure - Evaluation

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Secret Erasure - Evaluation

- Benchmark using MiBench (30x)

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Secret Erasure - Evaluation

- Benchmark using MiBench (30x)
- Function Based - **1.9 - 3.3x slower**

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Secret Erasure - Evaluation

- Benchmark using MiBench (30x)
- Function Based - **1.9 - 3.3x slower**
- Stack Based - **1 - 2x slower**

UNIVERSITÄT PADERBORN

Clang/LLVM Solutions

Secret Erasure - Evaluation

- Benchmark using MiBench (30x)
- Function Based - **1.9 - 3.3x slower**
- Stack Based - **1 - 2x slower**
- Call Graph Based - **Negligible**

UNIVERSITÄT PADERBORN

Future Work and Conclusion

- Compilers should protect implicit requirements

UNIVERSITÄT PADERBORN

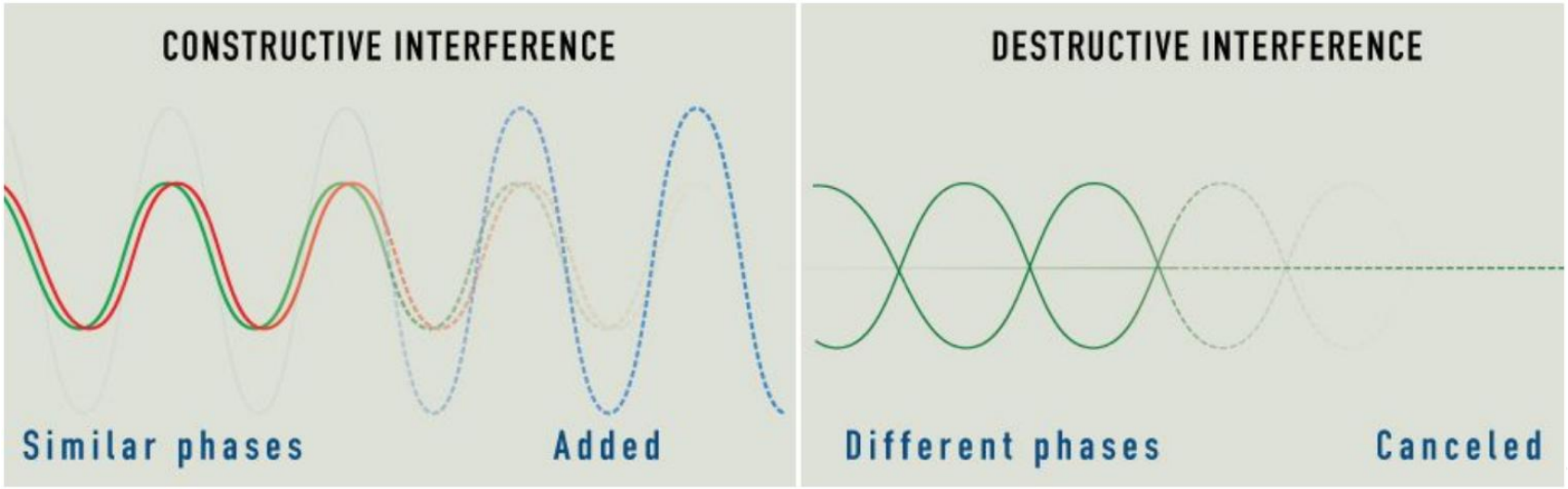
Future Work and Conclusion

- Compilers should protect implicit requirements
- Need mechanisms to convey assumptions

UNIVERSITÄT PADERBORN

Future Work and Conclusion

- Compilers should protect implicit requirements
- Need mechanisms to convey assumptions
- **Constructive**, not destructive interference between ***Compiler Developers*** and ***Security Engineers***



UNIVERSITÄT PADERBORN

References

Simon, Laurent and Chisnall, David and Anderson, Ross (2018)

What You Get Is What You C: Controlling Side Effects in Mainstream C Compilers

2018 IEEE European Symposium on Security and Privacy (EuroS&P)

Yang, Zhaomo and Johannesmeyer, Brian and Olesen, Anders Trier and Lerner, Sorin and Levchenko, Kirill (2017)

Dead Store Elimination (Still) Considered Harmful

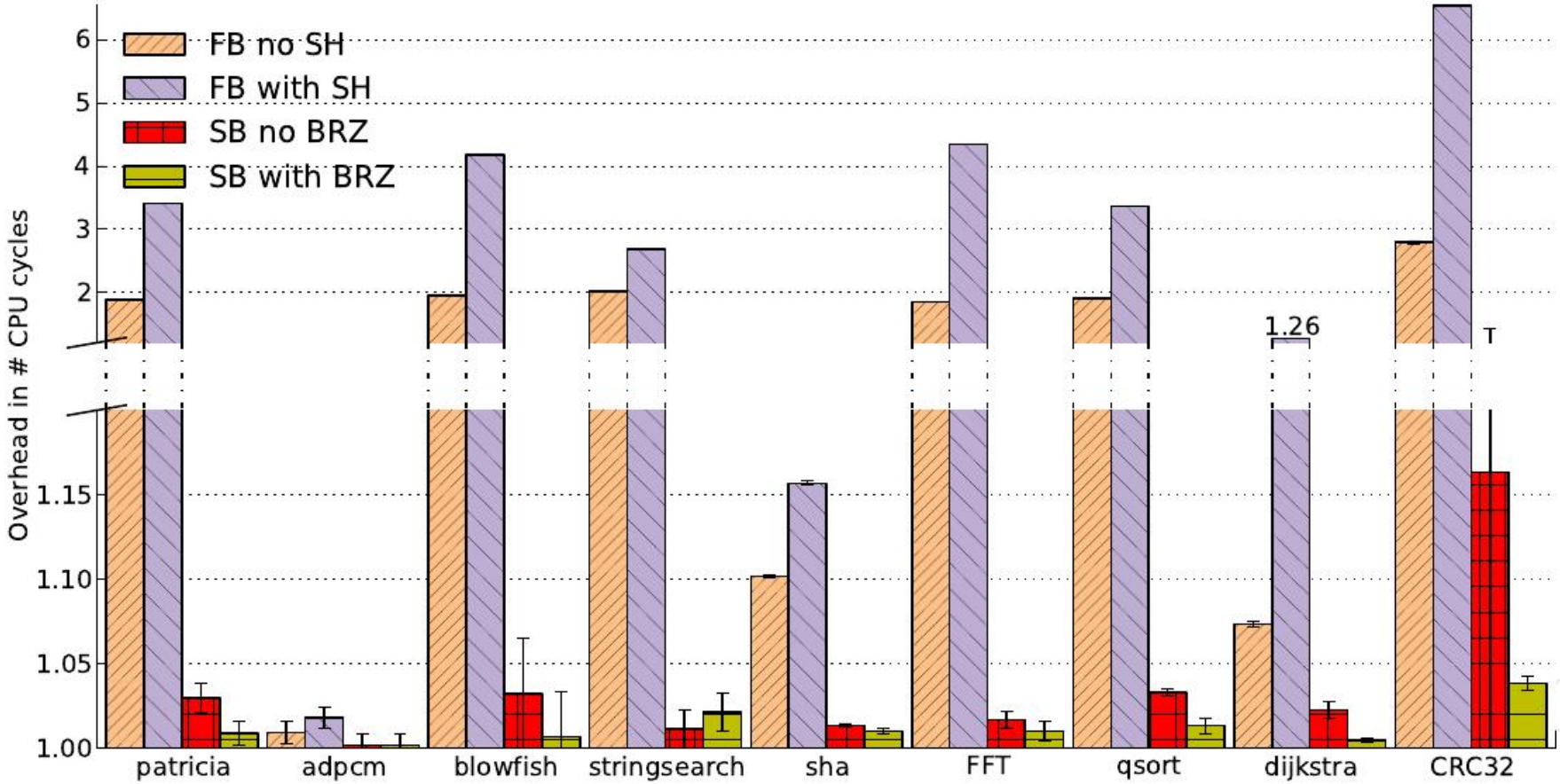
2017

Images

http://www.webexhibits.org/causesofcolor/15E.html

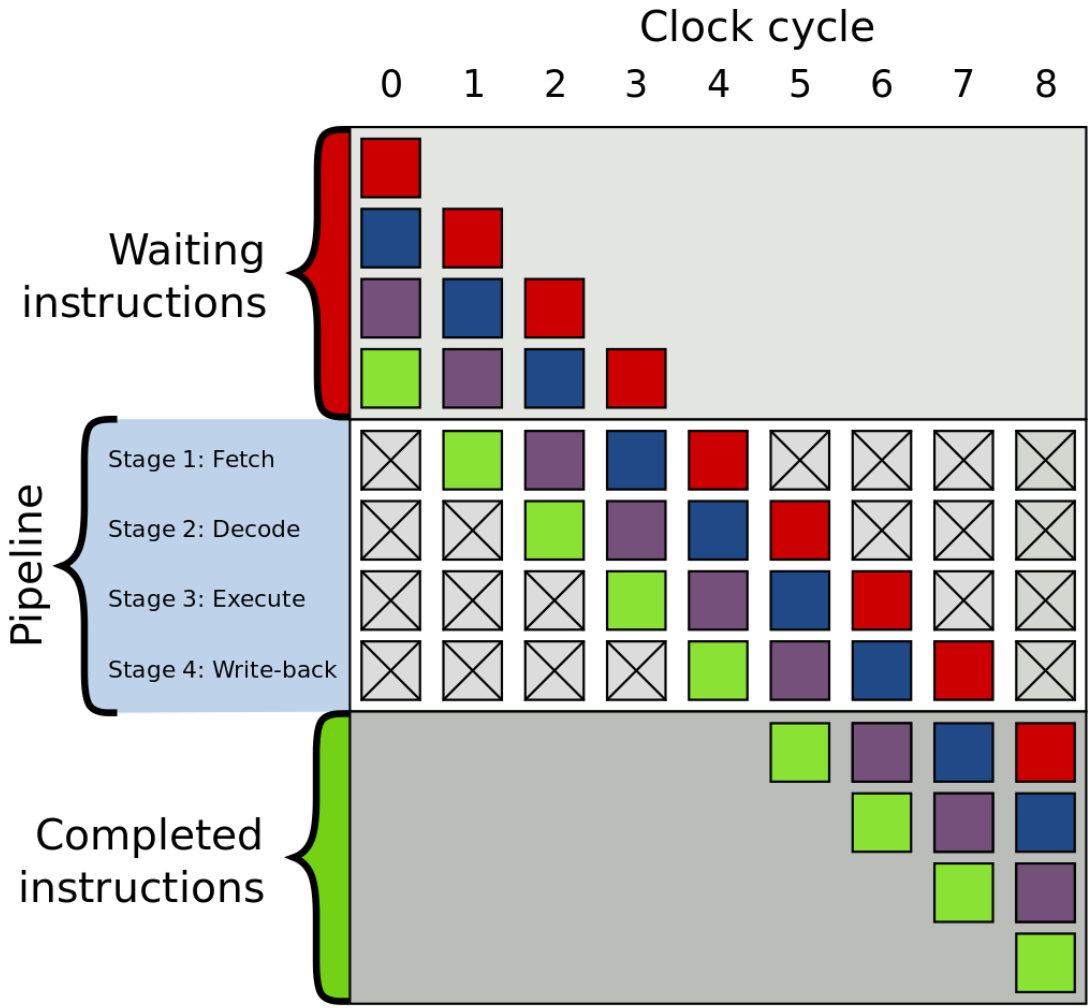
UNIVERSITÄT PADERBORN

Secret Erasure



UNIVERSITÄT PADERBORN

Branch Prediction



UNIVERSITÄT PADERBORN

CMOV

Reinitialize ECX to 0

```
1
2  XOR EBX, EBX ; Clear register for later
3  ADD ECX, [SMALL_COUNT] ; Adjusts by some counter value
4  JNC Continue ; If ECX didn't overflow, continue
5  MOV ECX, EBX ; Reinitialize ECX if it overflowed
6  Continue:
7
8  ; Using CMOV:
9
10 XOR EBX, EBX ; Clear register for later
11 ADD ECX, [SMALL_COUNT] ; Adjusts by some counter value
12 CMOVC ECX,EBX ; If ECX overflowed, reinitialize to EBX
```