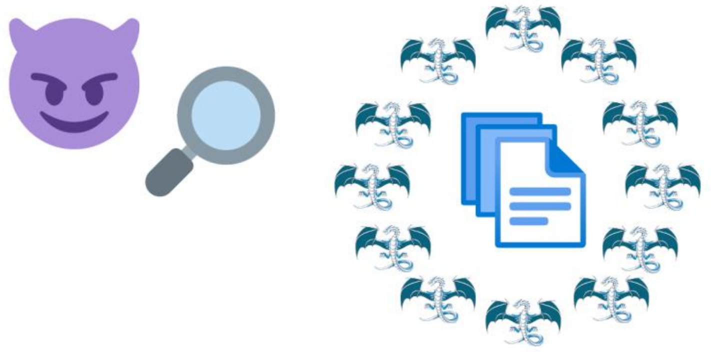
Hardening the Core: Challenges in Mitigating Hardware Vulnerabilities with LLVM



Reshabh K Sharma

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

This work was informed and inspired by collaborations with Michael Flanders (then at UW), Alexandra Michael (UW), Dan Grossman (UW), David Kohlbrenner (UW)

Safety-Critical Code Needs to be Hardened

Safety-critical code works with data that we do not want a malicious actor to infer.

Examples of Safety Critical Code:

- Password hash comparisons
- Cryptographic key equality checks
- Block cipher implementations
- Public key signature verification
- Message authentication code

Side Channel Attack are a Real Threat

Side-channel attacks work by leaking data through **unintended side effects** of program execution.

These side effects such as *timing, power usage, or cache behavior* can vary depending on secret-dependent parts of the code.

Example: if a password check returns faster when the password matches, that timing difference can reveal information to an attacker.

Constant Time Programming

Constant-time (CT) programming is a paradigm designed to ensure that the observable effects of code execution such as timing, memory access, or control flow remain **independent** of secret or sensitive inputs.

Properties of constant-time code:

- Avoid conditional branching on secret data
- Avoid secret-dependent memory access patterns
- Use constant-time arithmetic and operations
- Process all data uniformly

Is Constant Time Programming Enough?

Unfortunately, No!

Even if a program is written to avoid timing or control flow variations, the microarchitectural implementation such as caches, speculative execution, or prefetchers can still introduce observable differences that attackers exploit as side channels.

Is Constant Time Programming Enough?

Unfortunately, No!

Even if a program is written to avoid timing or control flow variations, the microarchitectural implementation such as caches, speculative execution, or prefetchers can still introduce observable differences that attackers exploit as side channels.

GoFetch

Hertzbleed

MELTDOWN

Microarchitectural Side Channel

Microarchitectural implementation details can cause *constant-time programs* to exhibit **exploitable side effects**.

Example:

- Speculative Execution
- Silent Store
- Computation Simplification
- Value Prediction

… and many more

Microarchitectural Side Channel

Microarchitectural implementation details can cause *constant-time programs* to exhibit **exploitable side effects**.

Example:

Speculative Execution

```
…
cmp rdi, rsi
cmovge rdi, rsi
mov al, byte [rcx + rdi]
…
```

Speculative execution can transiently execute mov al, byte [rcx + rdi] with out-of-bounds rdi before clamp takes effect.

Microarchitectural Side Channel

Microarchitectural implementation details can cause *constant-time programs* to exhibit **exploitable side effects**.

Example:

Silent Store

```
…
mov [esp], eax
mov [esp], ebx
…
```

One store gets silenced or optimized when EAX and EBX hold the same value.

Microarchitectural Side Channel

Microarchitectural implementation details can cause *constant-time programs* to exhibit **exploitable side effects**.

Example:

Computation Simplification

```
…
add eax, ebx
…
```

Computation gets optimized when EBX is known to be zero.

Microarchitectural Optimizations

Microarchitectural Optimizations (speculative execution, silent stores, computation simplification and many more) may result in creating side effects that can be observed by an attacker to leak secrets.

Take Away: The microarchitectural optimizations can be exploited and break the constant-time code.

Defending Against Hardware Vulnerabilities

Code Pattern => Triggers Microarchitectural Optimizations

(Potentially Vulnerable)

Defending Against Hardware Vulnerabilities

Code Pattern => Triggers Microarchitectural Optimizations

(Potentially Vulnerable)

Can we not turn off the hardware optimization completely in future hardware?

Defending Against Hardware Vulnerabilities

Code Pattern => Triggers Microarchitectural Optimizations

(Potentially Vulnerable)

Can we not turn off the hardware optimization completely at OS level?

Defending Against Hardware Vulnerabilities

Code Pattern => Triggers Microarchitectural Optimizations

(Potentially Vulnerable)

Can we not programatically turn off the hardware optimization selectively for code working with secrets?

Mitigation as Code Transformation

Code Pattern => Triggers Microarchitectural Optimizations

(Potentially Vulnerable)

Can we get rid of all the code patterns dealing with secrets that triggers the potentially vulnerable optimization?

Mitigation as Code Transformation

Code Pattern => Triggers Microarchitectural Optimizations

(Potentially Vulnerable)

- Zero Day Solution
- Works with older hardware
- Selectively works on code dealing with secrets

Mitigation as Code Transformation

Code Pattern => Triggers Microarchitectural Optimizations

(Potentially Vulnerable)

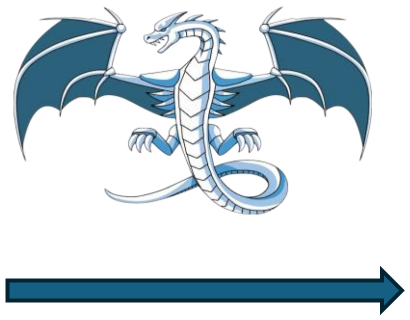
We can get rid of all the code patterns dealing with secrets that triggers the potentially vulnerable optimization.

Can we transform all the code patterns to ensure that they do not trigger that specific hardware optimization?

Mitigation as Code Transformation

Speculative Execution

```
…
cmp rdi, rsi
cmovge rdi, rsi
mov al, byte [rcx + rdi]
…
```



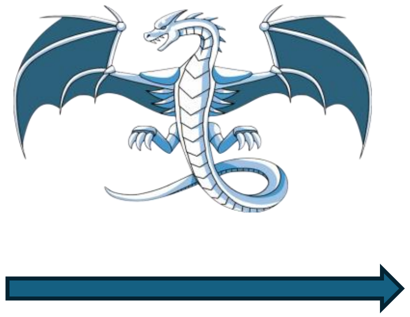
```
…
cmp rdi, rsi
cmovge rdi, rsi
lfence
mov al, byte [rcx + rdi]
…
```

Speculative execution can transiently execute mov al, byte [rcx + rdi] with out-of-bounds rdi before clamp takes effect but not with the fence.

Mitigation as Code Transformation

Silent Store

```
…
mov [esp], eax
mov [esp], ebx
…
```



```
…
mov [esp], eax
mov [esp], r11
mov [esp], ebx
…
```

With value of r11 such that it is never eax or ebx, we can ensure that silent store optimization never triggers for the ebx and r11 mov

Challenges in Mitigating inside a Compiler

What abstraction shall we implement them on?

- In LLVM IR
- In the backend before regalloc
- In the backend after regalloc

Example: In the backend, after regalloc each mitigation needs to proactively reserve registers to be able to use them for temporaries.

Challenges in Mitigating inside a Compiler

Where shall we schedule them in the pass pipeline?

At the very end so that no other optimization or pass can interfere with their working.

Challenges in Mitigating inside a Compiler

Where shall we schedule them in the pass pipeline?

At the very end so that no other optimization or pass can interfere with their working.

Who is at the end when everyone is at the end?

Optimizations vs Mitigations

Mitigations needs to be *applied for anything that can potentially trigger the hardware optimization* making the optimization criteria more conservative than a mitigation.

Optimizations vs Mitigations

Phase ordering is a known issue in optimizing compilers. *A suboptimal phase order may result in less performant code but is not unsafe*.

In contrast, mitigations must be handled carefully, as one mitigation undoing another can be unsafe or, worse, provide a false sense of security

Open Research Challenges

- How can the developer reason about the passes **ahead of their pass**?

- How can we automatically figure out **conflicting transformations**?

Handling Inline Assembly using Bolt

Inline assembly cannot be transformed by the compiler.

Bolt can be used to handle inline assembly, but not all transformation or analysis can be done at that abstraction.

Bolt as post transformation checker

Bolt can also be used as a post transformation checker to check if all the potential code triggers have been transformed.

Conclusion

Compiler-based defenses provide a **flexible and timely response** to *emerging threats*. They also enhance the overall security posture by layering defenses, protecting systems even when *hardware-based solutions fall short or are not feasible*.

The compiler infrastructure is well-suited for implementing mitigations as program transformations, but since it was never designed for this purpose, there are limitations we should address.