Broad-Based Side-Channel Defenses for Modern Processor Architectures

Ashay Rane
Advisors: Calvin Lin and Mohit Tiwari
The University of Texas at Austin
Sensitive Information in Various Applications

Applications in several domains operate on private or confidential information.

Cryptography  |  Web Browsers  |  Machine Learning

We want to prevent leakage of sensitive information
Many Techniques for Preventing Information Leakage

- Encryption
- Authentication + Authorization
- Sandboxing

user: jane
pass: *****
2FA: ******
Many Techniques for Preventing Information Leakage

- Virtual Machines or Containers
- Remote Execution
- Encryption
- Authentication + Authorization
- Sandboxing
But information can still leak through so-called side channels
Example of Side Channel Information Leakage

Jane, a grad student

user: jane
pass: *****
2FA: *****
Example of Side Channel Information Leakage

Jane, a grad student
Example of Side Channel: Rendering Characters

0

Rendered using lines and curves
Example of Side Channel: Rendering Characters

Rendered using lines and curves
Example of Side Channel: Rendering Characters

Rendered using lines and curves
Execution Time for Rendering Characters
Renderer’s execution time can reveal document contents even if the document is encrypted and protected by the strongest password and run inside an SGX Enclave / Container / Virtual Machine on a computer deep down in the ocean.
Real-World Attack on FreeType Renderer

[Xu et al., Oakland-2015]

Application converts document into a bitmap image

Application runs inside an SGX-like enclave

Malicious OS observes page faults

100% text recovered by OS
Memory Address Trace while Rendering Characters

Rendered Character:  X  Y  Z
Instruction Trace while Rendering Characters

Rendered Character:  
- X
- Y
- Z

Basic Block ID

0 100 200 300

Basic Block Execution (Time)

0 1000 2000 3000 4000 5000
Secrets may Leak Through Many Side Channels

Secret data can be inferred using many side channels

- **Application Program**: e.g. number of instructions
- **Operating System**: e.g. page faults
- **Microarchitecture**: e.g. branch predictor, cache, program counter
- **Physical Hardware**: e.g. power consumption
What is the **Core Vulnerability**?

The code executes different sequences of instructions for rendering each character.
Different input values execute different program paths, thus causing variation in execution.
**Prior Side Channel Defenses**

Are **point solutions**, since they **focus on symptoms** and not the root cause.

<table>
<thead>
<tr>
<th>Application Program</th>
<th>e.g. number of instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>e.g. page faults</td>
</tr>
<tr>
<td>Microarchitecture</td>
<td>e.g. branch predictor, cache, PC, DRAM addresses</td>
</tr>
<tr>
<td>Physical Hardware</td>
<td>e.g. power consumption</td>
</tr>
</tbody>
</table>

[ISCA07], [ISCA08], [HPCA09], [NDSS15], [CCS13a]
### Prior Side Channel Defenses

Are **point solutions**, since they **focus on symptoms** and not the root cause.

<table>
<thead>
<tr>
<th></th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Program</td>
<td>e.g. number of instructions</td>
</tr>
<tr>
<td>Operating System</td>
<td>e.g. page faults</td>
</tr>
<tr>
<td>Microarchitecture</td>
<td>e.g. branch predictor, cache, PC, <strong>DRAM addresses</strong> [ISCA13], [CCS13b], [CCS13c], [ASIACRYPT11]</td>
</tr>
<tr>
<td>Physical Hardware</td>
<td>e.g. power consumption</td>
</tr>
</tbody>
</table>
Prior Side Channel Defenses

Are **point solutions**, since they **focus on symptoms** and not the root cause

<table>
<thead>
<tr>
<th>Application Program</th>
<th>e.g. number of instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>e.g. page faults</td>
</tr>
<tr>
<td>Microarchitecture</td>
<td>e.g. branch predictor, cache, PC, DRAM addresses</td>
</tr>
<tr>
<td>Physical Hardware</td>
<td>e.g. <strong>power consumption</strong></td>
</tr>
</tbody>
</table>

[DATE04], [VLSID08], [SCS09], [ESSCIRC02], [FSE05], [ICICS06], [CHES12], [IEEE15], [IACR10]
Example of a **Point Solution**

GhostRider [ASPLOS-15]

Original Code

```plaintext
if (secret == 0) {
    x <- load ptr_1
    y <- load ptr_2
} else {
    z <- load ptr_3
}
```

Transformed Code

```plaintext
if (secret == 0) {
    x <- load ptr_1
    y <- load ptr_2
} else {
    z <- load ptr_3
    d <- load dummy
}
```

Ensure equal load instructions of each path
Example of a **Point Solution**

GhostRider [ASPLOS-15]

Optimizing compilers **may break the security guarantee**

Original Code

```cpp
if (secret == 0) {
    x <- load ptr_1
    y <- load ptr_2
} else {
    z <- load ptr_3
}
```

Transformed Code

```cpp
if (secret == 0) {
    x <- load ptr_1
    y <- load ptr_2
} else {
    z <- load ptr_3
    d <- load dummy
}
```

Ensure equal load instructions of each path

**Dead Code Elimination**
Example of a **Point Solution**

GhostRider [ASPLOS-15]

**Caches and prefetchers may break the security guarantee**

**Original Code**

```plaintext
if (secret == 0) {
    x <- load ptr_1
    y <- load ptr_2
} else {
    z <- load ptr_3
}
```

**Transformed Code**

```plaintext
if (secret == 0) {
    x <- load ptr_1
    y <- load ptr_2
} else {
    z <- load ptr_3
    d <- load dummy
}
```
Performance Impact of Using Point Solution

Disabled optimizations result in significant performance overhead

![Graph showing slowdown (X) for different operations with and without optimizations]
Prior Side Channel Defenses

Are **point solutions**, since they **focus on symptoms** and not the root cause, and they may not compose well.

Some require **disabling of optimizations** in compiler and microarchitecture.

Some **inflexible** because they **cannot be tailored to the program** or to portions of the program.
Simultaneously defends from a broad class of side channels (specifically, all digital side channels) using a single solution.

- **Application Program**
  - e.g. number of instructions

- **Microarchitecture**
  - e.g. branch predictor, cache, program counter

- **Physical Hardware**
  - e.g. power consumption, EM radiation
Simultaneously defends from a broad class of side channels (specifically, all digital side channels) using a single solution

Compatible with many optimizations in the compiler and in the microarchitecture (e.g. caches, prefetchers)

Built into a compiler, making our solution tailored to application programs
Our Solution

Program that **leaks info** via side channels

Annotations that mark secrets inputs

Compiler

Program that **does not leak info** via side channels

**Step 1 [Code Analysis]**: Identify portions of the program that need security.

**Step 2 [Code Transformation]**: Change relevant portions of the program, so that they don’t leak secrets through side channels.
Secret Input → Program → Side-Channel Observation
Key Approach of Our Solution

Secret #1, #2, #3, ...

Transformed Program

Only One Side-Channel Observation
CFG of Program \( P \) that **Leaks** Information via Program Counter

Program \( Q \) that **Does Not Leak** Information via Program Counter

Code transformation removes variations in executed basic blocks

Transformed program executes all instructions regardless of input value.
**Challenges in Eliminating Variations**

Naively executing all instructions will produce invalid results.

Incorrect Output | Crashing Execution | Stuck Program
Original Program

```java
if (secret > 5) {
    x = 13;
} else {
    x = 15;
}
```

Incorrect Transformation

```java
if (secret > 5) {
  
  x = 13;
}
```

```java
x = 13;
```

```java
x = 15;
```
Ensuring Correctness

Original Program

```java
if (secret > 5) {
    x = 13;
} else {
    x = 15;
}
```

Correct Transformation

```java
(secret > 5)  x = 13;
(secret <= 5) x = 15;
```
Key Building Block: **Predicated Write Operation**

Cond → Predicated Write Operation → Output

Output = a if cond = TRUE, b otherwise

**Implementation in x64 assembly:**

```
mov a -> output       // Set destination
test cond, cond      // Check if non-zero
cmovz b -> output    // Conditional update
test a, a             // Overwrite flags
```
Ensuring Correctness

Original Program

```java
if (secret > 5) {
    x = 13;
} else {
    x = 15;
}
```

Correct Transformation

```java
pred = secret > 5;
x = pred_write(pred, 13, x);
x = pred_write(!pred, 15, x);
```
Key Building Block: **Predicated Write Operation**

The `pred_write()` function:

- Has same sequence of instructions.
- Accesses zero memory locations.
- Consumes same number of processor cycles (verified empirically).

`pred_write()` conditionally updates a memory location without leaking the predicate through side channels.
Key Building Block: **Predicated Write Operation**

The `pred_write()` function:

- Has same sequence of instructions.
- Accesses zero memory locations.
- Consumes same number of processor cycles (verified empirically).

`pred_write()` conditionally updates a memory location without leaking the predicate through side channels.

We can now **execute arbitrary* instructions**, but we allow them to update memory contents only if the instruction is part of the correct execution path.

* System calls and library function calls are outside the scope of our compiler’s transformations, since the callee’s code cannot be changed.
Challenges in Eliminating Variations

**Correct Execution**
The transformed program produces the same output as the original program.

**Crash-Free Execution**
Executing dummy instructions in the transformed program should not crash the program.

**Progress of Execution**
Transformed program should not get stuck when executing dummy instructions.
Challenges in Eliminating Variations

Correct Execution
The transformed program produces the same output as the original program.

Crash-Free Execution
Executing dummy instructions in the transformed program should not crash the program.

Progress of Execution
Transformed program should not get stuck when executing dummy instructions.
But Predication may Crash the Program

\[
v = 0; \\
\textbf{if} \ (\text{secret}) \ {\{} \\
\quad v = 10; \\
\quad y = x / v; \\
{\}}
\]

\[
v = 0; \\
v = \text{pred\_write}(\text{secret}, 10, v); \\
y = \text{pred\_write}(\text{secret}, x/v, y);
\]
But Predication **may Crash the Program**

```
v = 0;
if (secret) {
    v = 10;
    y = x / v;
}
```

After transformation:
```
v = 0;
```
```
\boxed{v = \text{pred_write}(secret, 10, v);}
```
```
y = \text{pred_write}(secret, x/v, y);
```

If `secret` is false, `v` is not updated, hence `v` remains 0.
But Predication **may Crash the Program**

```c
v = 0;
if (secret) {
    v = 10;
    y = x / v;
}
```

```c
v = 0;
```

**After transformation**

```c
v = pred_write(secret, 10, v);
y = pred_write(secret, x/v, y);
```

If `secret` is false, `v` is not updated, hence `v` remains 0.

Division by zero exception causes program to terminate.
But Predication may Crash the Program

v = 0;
if (secret) {
    v = 10;
    y = x / v;
}

Our solution masks exceptions by covertly changing divisor value.

v = 0;
if (secret) {
    v = pred_write(secret, 10, v);
    y = pred_write(secret, x/v, y);
}

v = 0;
v = pred_write(secret, 10, v);
t = pred_write(v == 0, 1, v);
y = pred_write(secret, x/t, t);
Our solution masks exceptions by covertly changing divisor value.

Our solution assumes that the pre-transformation program does not throw uarch exceptions.
Challenges in Eliminating Variations

Correct Execution
The transformed program produces the same output as the original program.

Crash-Free Execution
Executing dummy instructions in the transformed program does not crash the program.

Progress of Execution
Transformed program should not get stuck when executing dummy instructions.
# Challenges in Eliminating Variations

<table>
<thead>
<tr>
<th>Correct Execution</th>
<th>Crash-Free Execution</th>
<th>Progress of Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>The transformed program produces the same output as the original program.</td>
<td>Executing dummy instructions in the transformed program does not crash the program.</td>
<td>Transformed program should not get stuck when executing dummy instructions.</td>
</tr>
</tbody>
</table>
But Predication may **Cause Infinite Loops**

Loops require a different transformation.
Transforming Loops

Assume $n$ is secret. Transformation should hide the number of executed iterations.
## Transforming Loops

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Transformed Code</th>
</tr>
</thead>
</table>
| `loop i :: 0 to n`  
  `x = x * y;`  
  `i = i + 1;` | `loop ctr :: 0 to C`  
  `ctr = ctr + 1;` |
Transforming Loops

Original Code

\[
\text{loop } i :: 0 \text{ to } n
\]
\[
x = x \times y;
\]
\[
i = i + 1;
\]

Transformed Code

\[
i = 0
\]
\[
\text{loop } \text{ctr} :: 0 \text{ to } C
\]
\[
x = \text{pred}_\text{write}(p, x \times y, x);
\]
\[
i = \text{pred}_\text{write}(p, i + 1, i);
\]
\[
\text{ctr} = \text{ctr} + 1;
\]

New predicate for loop body
Transforming Loops

Original Code

```
loop i :: 0 to n
    x = x * y;
    i = i + 1;
```

Transformed Code

```
i = 0
p = TRUE
loop ctr :: 0 to C
    x = pred_write(p, x * y, x);
    i = pred_write(p, i + 1, i);

    ctr = ctr + 1;
```
Transforming Loops

Original Code

```plaintext
loop i :: 0 to n
   x = x * y;
   i = i + 1;
```

Transformed Code

```plaintext
i = 0
p = TRUE
loop ctr :: 0 to C
   x = pred_write(p, x * y, x);
   i = pred_write(p, i + 1, i);
   p = pred_write(i == n, FALSE, p);
   ctr = ctr + 1;
```

New predicate for loop body

Turn predicate OFF to run dummy iterations.
Transforming Loops

Transformed Code

```plaintext
i = 0
p = TRUE
loop ctr :: 0 to C
    x = pred_write(p, x * y, x);
    i = pred_write(p, i + 1, i);
    p = pred_write(i == n, FALSE, p);
    ctr = ctr + 1;
```

Annotated by user, for example:
```
__max_loop_count(1024);
```

OR

Determined automatically using predictive mitigation [CCS’11]:
```
C = 1, 2, 4, 8, 16, 32, ...
```
Challenges in Eliminating Variations

**Correct Execution**
The transformed program produces the same output as the original program.

**Crash-Free Execution**
Executing dummy instructions in the transformed program does not crash the program.

**Progress of Execution**
Transformed program does not get stuck when executing dummy instructions.
CFG of Program $P$ that **Leaks** Information via Program Counter

Program $Q$ that **Does Not Leak** Information via Program Counter

Code transformation
### Other Variations in Program Execution

<table>
<thead>
<tr>
<th>Memory Addresses</th>
<th>Execution Time of Instructions</th>
<th>Coarse-Grain Power Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

59
Our Solution

Program that leaks info via relevant side channels

Which side channels exist on a given platform?

Compiler

Program that does not leak info via relevant side channels
Our Solution

Compiler Backend (e.g. LLVM codegen)

ISA (e.g. armv8, x64)

Microarchitecture (e.g. Gem5 Impl)

Source Code Analysis

Leakage Model

Compiler

Program that leaks info via relevant side channels

Program that does not leak info via relevant side channels

manual analysis
Leakage Model

Informs compiler about potential side-channel leakage due to IR-level instructions, for example:

- LLVM’s `div` instruction can leak information through **timing channel** on Intel Nehalem.
- LLVM’s `cmp` instruction can leak information through **power channel** on Gem5’s ARMv8 impl.
- LLVM’s `load` instruction can leak information through **cache channel** on Gem5’s RISC-V impl.
**Construction of Leakage Model**

We apply classical taint tracking to the microarchitecture and the ISA definition.

<table>
<thead>
<tr>
<th>Taint Sources (Private Information)</th>
<th>Taint Sinks (Public Observations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Registers</td>
<td>● Cycles (timing)</td>
</tr>
<tr>
<td>● Memory Locations</td>
<td>● Exceptions</td>
</tr>
<tr>
<td>● Operands</td>
<td>● Program Counter</td>
</tr>
<tr>
<td></td>
<td>● ...</td>
</tr>
</tbody>
</table>

Which instructions cause variations in taint sinks based on the taint source?
**Construction of Leakage Model**

Sample Implementation of `div` Instruction

```c
func exec_div(divisor) {
    dividend = reg_set[rax]

    while (dividend < divisor) {
        ...
        cycle_count += 1;
    }
}
```

Taint Sources:
- `divisor`, `reg_set`

Taint Sink:
- `cycle_count`
Construction of Leakage Model

Sample Implementation of \texttt{div} Instruction

\begin{verbatim}
func exec_div(divisor) {
    dividend = reg_set[rax]
    while (dividend < divisor) {
        ...
        cycle_count += 1;
    }
}
\end{verbatim}

Taint Sources: \texttt{divisor}, \texttt{reg\_set}

Taint Sink: \texttt{cycle\_count}
**Construction** of Leakage Model

Sample Implementation of `div` Instruction

```c
func exec_div(divisor) {
    dividend = reg_set[rax]
    while (dividend < divisor) {
        ... 
        cycle_count += 1;
    }
}
```

Taint Sources:
- `divisor`, `reg_set`

Taint Sink:
- `cycle_count`

The value of `cycle_count` depends on `dividend` and `divisor`. 
Our Solution

- **Compiler Backend** (e.g. LLVM codegen)
- **ISA** (e.g. armv8, x64)
- **Microarchitecture** (e.g. Gem5 Impl)

Source Code Analysis

Leakage Model

Compiler

Program that leaks info via relevant side channels

Program that does not leak info via relevant side channels

---

manual analysis
We would like to guarantee that critical programs do not contain side channels.

**Potential Approach #1**

Write formal proof of non-interference for each program and verify it using Coq / ACL2 / Dafny / ...

*Advantage:* Small specification  
*Disadvantage:* Time and effort intensive

**Potential Approach #2**

Extend classical taint tracking analysis and check for side channels.

*Advantage:* Cheap, easy to run  
*Disadvantage:* Large specification, hence more likely to contain bugs

**Can we obtain formal guarantees using a cheap analysis?**
Our Solution: Verified Leakage Analyzer

Program

Taint Tracking Analysis

Proof

Analysis Output

Specification (~200 lines)

Trustworthy Output

One-Time Verification
## Evaluation Programs

<table>
<thead>
<tr>
<th>GhostRider Benchmarks</th>
<th>Machine-Learning Apps</th>
<th>Cryptographic Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Heap Add</td>
<td>- Motion Tracking</td>
<td>- Lattice Crypto</td>
</tr>
<tr>
<td>- Dijkstra SSSP</td>
<td>- SVM Classifier</td>
<td>- Curve-25519</td>
</tr>
<tr>
<td>- Binary Search</td>
<td></td>
<td>- Poly-1305</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Top-k Search</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Bellman Ford</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- PageRank</td>
<td></td>
</tr>
</tbody>
</table>

|                      | 24 lines               | 1,152 lines            |
|                      | 50 lines               | 618 lines              |
|                      | 35 lines               | 363 lines              |
| Font Renderer        | 592 lines              |                        |
| Hash Table           | 534 lines              |                        |
| Bloom Filter         | 811 lines              |                        |
| Lattice Crypto       |                        |                        |
| Curve-25519          |                        |                        |
| Poly-1305            |                        |                        |

**GhostRider Benchmarks**
- Heap Add: 24 lines
- Dijkstra SSSP: 50 lines
- Binary Search: 35 lines

**Machine-Learning Apps**
- Motion Tracking: 2,903 lines
- SVM Classifier: 2,974 lines

**Cryptographic Programs**
- Lattice Crypto: 1,152 lines
- Curve-25519: 618 lines
- Poly-1305: 363 lines

**Utility Programs**
- Font Renderer: 592 lines
- Hash Table: 534 lines
- Bloom Filter: 811 lines

**Graph Kernels**
- Top-k Search: 96 lines
- Bellman Ford: 89 lines
- PageRank: 111 lines
Our Solution: **All Digital Side Channels**

![Chart showing slowdown for different applications and algorithms with Ghost Rider, Machine Learning, Graph Kernel, Utility, Crypto, and GEO MEAN categories.]
Our Solution: **Instruction Trace** Side Channel

![Graph showing slowdown for various tasks]
Our Solution: **Cache** Side Channel

![Graph showing slowdown for various tasks]

- **Ghost Rider**
- **Machine Learning**
- **Graph Kernel**
- **Utility**
- **Crypto**
- **GEO MEAN**
Our Solution: **Timing** Side Channel
Broad **Research Goals**

How can we **identify** whether a given ISA + Microarchitecture leaks side channels?

How can we **transform** programs so that they do not leak information through side channels?

How can we **validate** that programs do not leak information through side channels?
Future Research in Computer Architecture

Problem: Existing compilers discard a wealth of information from the binary, while microarchitectures try to reconstruct the same info from dynamic analysis.

Examples:
- Prefetchers reconstructing streams
- Branch predictors learning biases
- Value prediction

Research Ideas:
- Extend the ISA to include more information than functional behavior.
- Concurrent program analysis in microarchitecture (similar to dynamic languages).
Future Research in Computer Security

Problem: Optimizations need careful evaluation to ensure that they don’t break security guarantees of side-channel defenses.

Example: Several common case optimizations introduce timing channel leakage.

Research Ideas:
- Formalize compiler and microarchitecture optimizations so that they can be analyzed
- Construct new compiler and microarchitectural optimizations that are inherently compatible with side-channel defenses.
Conclusion

1. Side channels are an important problem and they are hard to close.

2. Although several point solutions exist, they may not compose, so we need broad-based defenses.

3. By extending program analyses to the hardware design, we can design more effective and efficient defenses.
Thanks to My Collaborators and Sponsors

Calvin Lin, UT Austin
Mohit Tiwari, UT Austin
K Rustan M. Leino, Microsoft Research
Chris Hawblitzel, Microsoft Research
Laure Thompson, Cornell University
Joshua Eversmann, Civitas Learning
Greg McDonald, HBK Investments

Bryan Parno, Carnegie Mellon University
Jacob R. Lorch, Microsoft Research
Srinath Setty, Microsoft Research
Manos Kapritsos, University of Michigan
Barry Bond, Microsoft Research
Raymond Chee, Carnegie Mellon University
Varun Adiga, UT Austin