Side-Channel Defenses for Modern Processor Architectures

Ashay Rane
Advisors: Calvin Lin and Mohit Tiwari
The University of Texas at Austin
Secret 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 secret information*
Many Techniques for Preventing Information Leakage

Virtual Machines and Containers for isolating applications from each other.

user: jane
pass: *****
But these techniques do not prevent information leakage through so-called side channels
Example of Side Channel Information Leakage

Jane, a grad student

Malicious Cloud Provider
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

V

Rendered using lines and curves
Execution Time for Rendering Characters

![Graph showing execution time for rendering characters]
**Execution Time** for Rendering Words

Words of same length take different times to render

<table>
<thead>
<tr>
<th>Word</th>
<th>Rendering Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>tranquilizer</td>
<td>$734 \times 10^3$ cycles</td>
</tr>
<tr>
<td>convincingly</td>
<td>$809 \times 10^3$ cycles</td>
</tr>
<tr>
<td>experiencing</td>
<td>$853 \times 10^3$ cycles</td>
</tr>
<tr>
<td>demagnetized</td>
<td>$943 \times 10^3$ cycles</td>
</tr>
</tbody>
</table>
Execution time of rendering process can reveal information about document even if the document is encrypted.
Memory Address Trace while Rendering Characters

Rendered Character:  
- X
- Y
- Z

[Graph showing memory access over time with X, Y, and Z tracked]
Instruction Trace while Rendering Characters

Rendered Character:  
- X
- Y
- Z

Basic Block ID

Basic Block Execution (Time)
Secrets may Leak Through **Many Side Channels**

<table>
<thead>
<tr>
<th>Component</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application Program</strong></td>
<td>e.g. execution time</td>
</tr>
<tr>
<td><strong>Operating System</strong></td>
<td>e.g. page faults</td>
</tr>
<tr>
<td><strong>Microarchitecture</strong></td>
<td>e.g. branch predictor, cache, program counter</td>
</tr>
<tr>
<td><strong>Physical Hardware</strong></td>
<td>e.g. power consumption</td>
</tr>
</tbody>
</table>

Secret data can be inferred using many side channels
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. <strong>execution time</strong></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>

[ISCA12], [ASPLOS15], [CHES00], [ICISC03], [ICISC05], [ICISC10]
## Prior Side Channel Defenses

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

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

[ISCA13], [CCS13b], [CCS13c], [ASIACRYPT11]
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. execution time</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>
</tbody>
</table>
| Physical Hardware   | e.g. **power consumption** [
|                     | [DATE04], [VLSID08], [SCS09], [ESSCIRC02], [FSE05], [ICICS06], [CHES12], [IEEE15], [IACR10]] |
Example of a **Point Solution**

GhostRider [ASPLOS-15]

Original Code

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

Transformed Code

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

Ensure equal running time of each path
Example of a **Point Solution**

GhostRider [ASPLOS-15]

Optimizing compilers 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
}
```

---

Ensure equal running time of each path

---

Dead Code Elimination
Example of a Point Solution

GhostRider [ASPLOS-15]

Caches and prefetchers may break the security guarantee

Original Code

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

Transformed Code

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

Ensure equal running time of each path

- Cache hit
- Cache hit
- Cache hit
- Miss!
Performance Impact of Using Point Solution

Disabled optimizations result in significant performance overhead
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.
**Our Solution**

Simultaneously defends from many side channels using a single solution

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

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

Program that **leaks info** via side channels

Annotations that mark secrets values

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.
Key Ideas of Our Solution

Side channels exist due to variations in program execution. Thus, close side channels by removing variations in execution.

If we eliminate variations at the program level, then we can fix the root cause of the problem and close many side channels.
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
**Incorrect Transformation**

<table>
<thead>
<tr>
<th>Original Program</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (secret &gt; 5) {</code></td>
</tr>
<tr>
<td><code>x = 13;</code></td>
</tr>
<tr>
<td><code>} else {</code></td>
</tr>
<tr>
<td><code>x = 15;</code></td>
</tr>
<tr>
<td><code>}</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Incorrect Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (secret &gt; 5) { }</code></td>
</tr>
<tr>
<td><code>x = 13;</code></td>
</tr>
<tr>
<td><code>x = 15;</code></td>
</tr>
</tbody>
</table>
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**
- **a**
- **b**

**Predicated Write Operation**

Output:
- \( a \) if \( \text{cond} = \text{TRUE} \)
- \( b \) otherwise

Code:

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

After transformation

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

But Predication may Crash the Program

After transformation

v = 0;

\textcolor{red}{v = \text{pred\_write}(secret, 10, v);}

\textcolor{red}{y = \text{pred\_write}(secret, x/v, y);}

If \textit{secret} is false, \textit{v} is not updated, hence \textit{v} remains 0.
But Predication may Crash the Program

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

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

But Predication **may Crash the Program**

Our solution masks exceptions by covertly changing divisor value.

After transformation

v = 0;
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);
But Predication **may Crash the Program**

Our solution masks exceptions by covertly changing divisor value.

Our solution assumes that the pre-transformation program does not throw 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.
Assume \( n \) is secret. Transformation should hide the number of executed iterations.

\[
\begin{align*}
\text{loop } i : & : 0 \to n \\
x & = x \times y; \\
i & = i + 1;
\end{align*}
\]
Transforming Loops

Original Code

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

Transformed Code

```plaintext
loop ctr :: 0 to C
    ctr = ctr + 1;
```
Transforming Loops

Original Code

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

Transformed Code

\[
i = 0
\]

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

\[
ctr = ctr + 1;
\]
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);

ctr = ctr + 1;
```
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);
    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**

```
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

<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 does not get stuck when executing dummy instructions.</td>
</tr>
</tbody>
</table>
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>Power Consumption</th>
</tr>
</thead>
</table>
Variations in **Memory Address Trace**

result = table[secret];

addr := base(table) + secret
result := read addr

An adversary that can observe address, can also derive secret.

secret = addr - base(table)
Eliminating Variations in Memory Address Trace

Solution #1: Array Streaming

Accesses the entire array to read one element of the array.
Expensive to access entire array, but vector instructions, caches, and prefetchers reduce latency.

Solution #2: Software ORAM

Software version of Path ORAM [ccs’ 13], which shuffles memory to hide location of data.
<table>
<thead>
<tr>
<th>Memory Addresses</th>
<th>Execution Time of Instructions</th>
<th>Power Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Variable-Latency Floating-Point Instructions

Latency of Square-Root Instruction for Different Operand Types

- Normal: 11 cycles
- Not-a-Number: 7 cycles
- Zero: 7 cycles
- Infinity: 7 cycles
- Subnormal: >20x slower

Processor Cycles

Type of Operand

Normal | Not-A-Number | Zero | Infinity | Subnormal
0 | 7 | 7 | 153 |
Non-Secure Execution

A * B
(intended operation)
[next instr.]

time

C * D
(intended operation)
[next instr.]

time
Secure Execution

Spare SIMD lanes in SSE, SSE2 regs

A * B
(intended operation)

P * Q
(dummy operation)

C * D
(intended operation)

P * Q
(dummy operation)

Subnormal Operands

[next instr.]
Power Side Channel

Energy (μJ)

A B C D E F G H I J K L 1 2 3 4 5 6
Power Side Channel Defense

Transformed Application

Assembly Instructions

Instruction Set Architecture

Model of Microarchitecture Behavior

Microarchitecture

Hardware (Transistors)
Power Side Channel Defense

Transformed Application

Assembly Instructions

Instruction Set Architecture

Model of Microarchitecture Behavior

Microarchitecture

Power Model (e.g. McPAT)

Hardware (Transistors)
# Evaluation Programs

## Benchmarks from Related Work (GhostRider)

- **Heap Insertion**: 24 lines of code
- **Dijkstra-SSSP**: 50 lines of code
- **Binary Search**: 35 lines of code

## Applications

- **LibSVM Classifier**: 2,974 lines of code
- **Unicode Font Renderer**: 2,928 lines of code
- **Bloom Filter Library**: 811 lines of code
- **Top-k Selection**: 96 lines of code
- **Hash Implementation**: 534 lines of code

## Floating-Point Math Functions

- **Floating-Point Math Functions from the Musl C Library (e.g. sin, cos, log, etc.)**: 22,000+ lines of code
Performance: **GhostRider** (Related Work)

**GhostRider Benchmarks**
- Heap-Add
- Dijkstra-SSSP
- Binary-Search
- JWHash
- LibSVM
- Top-K
- Font-Renderer
- Bloom-Filter
- sin
- cos
- tan
- log
- exp
- GEO-MEAN

**Applications**
- 26
- 1987
- 112
- X
- X
- X
- X
- X
- X
- X
- X
- 180

**FP Math Functions**
Our Solution: **Instruction Trace** Side Channel

- GhostRider Benchmarks
- FP Math Functions
- Applications

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Slowdown (X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap-Add</td>
<td>2</td>
</tr>
<tr>
<td>Dijkstra-SSSP</td>
<td>4</td>
</tr>
<tr>
<td>Binary-Search</td>
<td>1</td>
</tr>
<tr>
<td>JW-Hash</td>
<td>2</td>
</tr>
<tr>
<td>LibSVM</td>
<td>2</td>
</tr>
<tr>
<td>Top-K</td>
<td>3</td>
</tr>
<tr>
<td>Font-Renderer</td>
<td>2</td>
</tr>
<tr>
<td>Bloom-Filter</td>
<td>1</td>
</tr>
<tr>
<td>Sin</td>
<td>4</td>
</tr>
<tr>
<td>Cos</td>
<td>4</td>
</tr>
<tr>
<td>Tan</td>
<td>3</td>
</tr>
<tr>
<td>Log</td>
<td>3</td>
</tr>
<tr>
<td>Exp</td>
<td>14</td>
</tr>
<tr>
<td>GEO-MEAN</td>
<td>3</td>
</tr>
</tbody>
</table>
Our Solution: **Memory Address Trace Side Channel**

- GhostRider Benchmarks
- Applications
- FP Math Functions

Bar chart showing slowdown (X) for various benchmarks and functions.

- Heap-Add: 2
- Dijkstra-SSSP: 6
- Binary-Search: 90
- jwHash: 2
- LibSVM: 2
- Top-K: 21
- Font-Renderer: 647
- Bloom-Filter: 1891
- sin: 4
- cos: 4
- tan: 5
- log: 3
- exp: 14
- GEO-MEAN: 13
Our Solution: **Timing** Side Channel

**GhostRider Benchmarks**

- Heap-Add: 2
- Dijkstra-SSSP: 6
- Binary-Search: 15
- JW-Hash: 2
- LibSVM: 2
- Top-K: 20
- Font-Renderer: 8
- Bloom-Filter: 16
- Sin: 273
- Cos: 310
- Tan: 227
- Log: 86
- Exp: 78
- GEO-MEAN: 22

**Applications**
## Security: Instruction Trace Side Channel

### MD5 checksum of Instruction Trace for LibSVM Machine-Learning Classifier

<table>
<thead>
<tr>
<th>Secret Input (Type of Data)</th>
<th>Before Transformation (Non-Secure Execution)</th>
<th>After Transformation (Secure Execution)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP Sweep</td>
<td>0a7391f6e51727cab1ed72e4e59c8639</td>
<td>84c18d083ec6249dd9cd3e649557ca67</td>
</tr>
<tr>
<td>Neptune</td>
<td>abae59da172c84458e6492b664cd175c</td>
<td>84c18d083ec6249dd9cd3e649557ca67</td>
</tr>
<tr>
<td>Benign</td>
<td>266ad6c4491f57583d8cb1c087a1d9b6</td>
<td>84c18d083ec6249dd9cd3e649557ca67</td>
</tr>
</tbody>
</table>
Security: **Power** Side Channel

Analog values like power consumption are not discrete

Hence we model a machine-learning adversary

Effective defense if classifier’s accuracy is no better than random chance.
Strengths of Our Approach

1. **Simultaneously Closes a Broad Class of Side Channels**
   By fixing root cause of the problem, we can close many side channels.

2. **Allows Optimization of Secure Code**
   Small runtime TCB permits several optimization on secure code.

3. **Closes or Mitigates Digital as well as Analog Side Channels**
   Our solution devises defenses by analyzing programs, models of the microarchitecture, and models of the physical hardware.
Limitations of Our Approach

1. **Cannot transform library calls and system calls**
   Source code for library functions and system calls is unavailable to the compiler for transformation. However, binary analysis techniques may be useful.

2. **Power channel security guarantee depends on the model**
   The ability of our compiler-based solutions to close the power side channels rests upon the level of detail of the power model.

3. **Room for performance improvement**
   We are currently exploring more aggressive compiler optimizations and microarchitectural modifications for reducing impact on performance.
Conclusion

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

2. Our solution simultaneously closes a broad class of side channels by eliminating or minimizing source-level variations in program execution.

3. Our solution can bridge the gap between digital techniques and power side channel by augmenting the solution with a power model.
Future Work

1. Reduce the performance impact of our solutions.

2. Formalize the code transformation in a program verification framework to prove the correctness of the transformations.
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

Bryan Parno, Carnegie Mellon University
Jacob R. Lorch, Microsoft Research
Srinath Setty, Microsoft Research
Manos Kapritsos, University of Michigan
Barry Bond, Microsoft Research