Rethinking Instruction Set Guarantees for Software-Driven Hardware Security

Ashay Rane    Calvin Lin    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: ******
Example of Side Channel Information Leakage

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](image-url)
### 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 in transit.
Memory Address Trace while Rendering Characters

Rendered Character:  X  Y  Z

Memory Location

Memory Accesses (Time)
Secrets may Leak Through Many Side Channels

Secret data can be inferred using many side channels

<table>
<thead>
<tr>
<th>Application Program</th>
<th>e.g. instruction count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>e.g. page faults</td>
</tr>
<tr>
<td>Microarchitecture</td>
<td>e.g. cache, branch predictor</td>
</tr>
<tr>
<td>Physical Hardware</td>
<td>e.g. EM radiation</td>
</tr>
</tbody>
</table>
Secrets may Leak Through Many Side Channels

Secret data can be inferred using many side channels

- Application Program
  - e.g. instruction count

- Operating System
  - e.g. page faults

- Microarchitecture
  - e.g. cache, branch pred

- Physical Hardware
  - e.g. EM radiation -- Non-Digital Side Channels

Digital Side Channels
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 Digital 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>
</tbody>
</table>

[ASPLOS15], [ICISC05], [ICISC10]
## Prior Digital 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>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th><strong>Application Program</strong></th>
<th>e.g. number of instructions</th>
</tr>
</thead>
<tbody>
<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, PC, <strong>DRAM addresses</strong></td>
</tr>
</tbody>
</table>

[ISCA13], [CCS13b], [CCS13c], [ASIACRYPT11]
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 digital side channels

Annotations that mark secrets values

Compiler

Program that **does not leak info** via digital 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.
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**

Original Program

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

Incorrect Transformation

```java
if (secret > 5) { }

x = 13;

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**

Predicated Write Operation:

- **cond**
- **a**
- **b**
- **output**

**Output**:
- a if cond = TRUE
- b otherwise

**Assembly 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;
if (secret > 5) {
    x = pred_write(pred, 13, x);
} else {
    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.
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.
Assume $n$ is secret. Transformation should hide the number of executed iterations.

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

**Our Solution’s Approach**

- **Actual** Iterations
  - (pred = true)

- **Dummy** Iterations
  - (pred = false)
result = table[secret];

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

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

secret = addr - base(table)

Solution #1: Array Streaming

Accesses the entire array to read one element of the array.

Solution #2: Software ORAM

Software version of Path ORAM [CCS’13], which shuffles memory to hide location of data.
Variable-Latency Floating-Point Instructions

Latency of Square-Root Instruction for Different Operand Types

Processor Cycles

Type of Operand

Normal Not-A-Number Zero Infinity Subnormal

< 20x slower

11 7 7 7 153
Our Approach: Model-Driven Compilation

- **Conditional Branch**
- **cmov** on registers

Program that **leaks info** via digital side channels

Program that **does not leak info** via digital side channels
Constructing a Model of Digital Side Channel Leakage

- Model constructed using code analysis (information flow analysis) of the ISA definition.

- **Sources**: Register and memory operands.
  **Sinks**: Digital side channel observations (e.g. instruction pointer, address trace, etc.).

- Any information flow between *sources* and *sinks* indicates the existence of a digital side channel.

- In our implementation, we analyze x86 and ARMv8 ISAs to construct the model.
Benefits of the Model-Driven Approach

1. **Compilers can generate code that is free from digital side channels**

   Enables automated flexible defenses since the compiler can selectively apply mitigation techniques only where needed.

   We have successfully transformed graph kernels, floating-point libraries, and end-user applications (like machine-learning classifiers) [Rane15, Rane16].

2. **Allows formal verification of the absence of digital side channels**

   We formally verified a handful of common cryptographic implementations to be free from digital side channels [Bond17].

   Eliminates the compiler from the trusted computing base.
Adding Security Contract to ISA

Existing ISAs provide a functional contract: \( \text{add } a, b, r \Rightarrow r = (a+b) \mod 2^{32} \)

We want to add security properties to ISA, e.g. \text{add} should consume fixed latency regardless of the operand values \( (a \text{ and } b) \), thus closing the timing side channel.

[ARM v8.4 allows setting a bit which ensures fixed timing of certain instructions]

Other security properties in ISA:
- Cache that exposes a portion of its space for a user-controlled scratchpad
- Memory controller that selectively enables ORAM accesses
Our goal is to let the compiler orchestrate the use of hardware security components.
1. Side channels are an important problem and they are hard to close.

2. The key to effective side channel defenses is a model of the microarchitectural side channel leakage.

3. We can build high-assurance and high-performance defenses by augmenting the ISA with security properties.