Detecting Side Channels at the ISA Level with Status Analysis Techniques

Raymond Chee

May 16, 2018

Abstract

Side channels attacks can steal sensitive information from programs and exploit hardware behavior to bypass traditional security measures. These side channels often come from inherent flaws in a computer system’s design rather than individual errors in a system’s implementation. Conducting extensive security reviews over a system by hand can be costly and time-consuming. This paper presents a system to automatically detect potential side channels in an instruction set architecture (ISA) specification, drastically reducing the search space a human needs to investigate. Although our analysis is limited to the ISA, the methodology we present in this paper can be applied to discover side channels in the microarchitecture. Our analysis finds six different information leaks that can occur at the ISA level.

1 Introduction

Programmers often develop software with a variety of security assumptions. Operating systems attempt to provide isolation between processes and provide access control mechanisms to prevent unprivileged programs from accessing restricted resources. Additionally, techniques like encryption give programmers the illusion that their data cannot be read by outsiders. However, in the presence of side channels, these assumptions do not hold. Side channels leak information from running programs, and researchers have demonstrated side channel attacks to break encryption schemes like AES[1] and RSA[2], identify private browsing activity on smartphones[3], and read privileged kernel memory[4, 5].

Whenever the hardware executes code, it produces various physical artifacts that may differ based on the data. By measuring these artifacts, an outsider can infer information about the values of the data used in the program. Even with OS isolation mechanisms and encryption in place, this inference is still effective. For instance, if a program performs an expensive operation when an input is even and nothing otherwise, an attacker can measure the program’s execution time and determine the parity of the input. These measurable artifacts are known as
side channels. Other examples of side channels include power consumption[1, 6], memory access latency[2], and electromagnetic radiation[7].

Much of the previous research on mitigating side channels focus on point solutions[8, 9, 10, 11] which seek to stop specific side channel attacks against individual implementations of algorithms while preserving the algorithm’s correctness. Although these approaches are helpful, they may miss other side channels, and they are difficult to scale since researchers must manually analyze every new implementation separately. Another approach to side channel defenses takes a broader view of the problem. Raccoon[12] and Escort[13] introduce compiler transformations which emit machine instructions designed to completely eliminate entire classes of side channels. This technique is agnostic to the source code of the program and could be a promising method to prevent side channels at scale. However, for these compiler transformations to be effective, they require a comprehensive model about the side channels present at the machine instruction level. Our work creates this model by introducing a methodology to determine if an ISA definition leaks information through side channels. Figure 1 shows how the analysis tool we present integrates with these compiler-based solutions.

We present a framework to analyze an ISA for potential side channels. This framework leverages an existing ISA specification language, Sail[14], and adds a layer to its compiler to enable data-flow analysis. We define data-flow analysis rules to determine if an instruction could leak information. Using this tool, we explore the ARMv8, CHERI, MIPS, Power, RISC-V, and x86 ISAs. Our analysis can be extended to other ISAs given a definition of their instruction semantics in Sail.

Our approach makes several key contributions. First, it introduces a methodology to detect side channels at the machine instruction level. Next, it creates a model of the ISA that a side channel aware compiler can use to treat each machine instruction appropriately. It allows architecture designers to determine if their ISAs have side channel vulnerabilities during prototyping, reducing the likelihood of including design flaws which leak side channels. Finally, we identify six potential side channels in various ISAs.

2 Threat Model

In our threat model, we consider two programs running on the same machine. These programs can be different browser tabs whose contexts should be inaccessible to each other. They can also be tenants in virtual machines sharing the same underlying hardware. We assume that in any architecture, the processor is a sealed chip which is physically inaccessible from the outside. Users run trusted code on the machine, but attackers can run concurrent programs which can determine various run-time statistics about the victim program. Additionally, since we are concerned with side channels at the ISA level, we conservatively assume that attackers are aware of the instructions the victim program is running.
Figure 1: Our analysis tool creates a model of the side channels in an ISA from the ISA’s specification. This model is used by a side channel aware compiler to produce safe executables.

Practically, our threat model assumes the attacker has the following capabilities:

- Attackers can observe whether or not memory was accessed. One way an attacker can obtain this information is through cache side channels\cite{15, 2}. However, we assume the actual contents of memory remain secret since the data can be encrypted.

- Attackers can detect whether an instruction causes a fault, which may cause the running program to terminate. This termination may be visible to the adversary through a broken network connection or a crashed process.

- Attackers are aware of the exact machine instruction the victim process is executing, including the operands that are used.

Given this threat model, we assume that users have secret data stored within the CPU’s general-purpose registers and main memory. Attackers aim to extract information about the data within these registers, but they can only do so by measuring the number of memory accesses and faults as a result of executing an individual instruction. Side channels which are created by executing multiple instructions are out of the scope of this analysis. These side channels can be closed by other methods such as the compiler-based defenses we discuss in Section 3.1.
3 Related Work

3.1 Compiler-Based Side Channel Defenses

Several previous papers present systems to automatically close side channels during compilation. Raccoon[12] is such a system which obfuscates the compiled binary, ensuring that the same instructions are run regardless of the program’s input. This approach is effective at preventing attacks that can determine the differences between the execution of unique code paths at the machine instruction level. However, if instructions exhibit data-dependent behavior, Raccoon cannot protect code from those side channels.

Another system, Escort[13], handles floating-point instructions which exhibit data-dependent latency. Escort normalizes floating-point instruction latency by grouping potentially problematic operations with slow dummy operations to guarantee a constant execution time.

These compiler-based approaches can use the ISA model we create to avoid introducing new side channels in the outputted binaries. For instance, if an instruction is known to conditionally raise an exception based on the value of some secret register data, the compiler can choose to emit a different set of side channel free instructions which accomplish the same result.

3.2 Microarchitectural Side Channels

The microarchitecture consists of the functional units of the processor, including the cores, caches, and memory controller. Szefer[16] identifies the characteristics of microarchitectures that expose side channels, which we summarize below.

1. Different instructions take different amounts of time to execute, exposing a timing channel.

2. Shared hardware leads to contention, allowing a co-resident processes to gain information about each other’s activity based on varying latencies when using the shared hardware.

3. Instructions can modify microarchitectural state, which is exposed to co-resident processes through the results and timings of their own operations.

4. During the execution of an instruction, the physical properties (like heat) of the processor changes in ways that sensors in the chip can measure.

The microarchitecture is a layer below the ISA, and an ISA can be implemented with a variety of microarchitectures. Thus, our analysis does not detect side channels at the microarchitecture level. However, we can apply the same techniques we use on the ISA to identify certain classes of side channels present in the microarchitecture. In particular, a functional specification of a microarchitecture will describe the state which is modified by each instruction, allowing an analysis to detect all side channels caused by point 3 in the characterization above. Additionally, if the specification includes information about the timings
or the physical side effects of its operations, this analysis can extend to detect side channels resulting from all four of the points above.

4 Background

4.1 Sail

We implement our analysis over specifications defined in Sail. Sail is a machine-parsable language built to define specifications of ISA semantics in processors. By providing an expressive type system, Sail enforces that definition writers provide precise specifications of their ISAs. The Sail compiler converts Sail ISA definitions to an intermediate language called Lem[17], which has a variety of features. The Lem toolchain includes compilers to generate both executable OCaml and Isabelle theorem prover code, as well as an interpreter to simulate the processor. It is theoretically possible to use any of Lem’s outputs to conduct the same analysis as we do, but we target the original Sail code in our analysis to simplify the process of manually verifying our results.

Functions in Sail. Function declarations in Sail contain annotations which inform our analysis. By convention, instructions are defined in functions named execute, which accept the instruction’s operands as arguments. Like any programming language, these functions can call other functions to implement the instruction semantics. Sail functions can be recursive, but they must be explicitly labeled as such. Functions are also labeled with their side effects, which we use in our analysis to determine if an instruction leaks information through side channels. Side effects include accesses to registers and memory, memory barrier invocations, and faults. In contrast, pure functions do not affect the physical state of the processor.

Data Types. Sail has a type system with several predefined types. We divide these types into two broad categories: scalar and container.

Scalar types can be thought of as individual units of data. These include bits, which store either a 0 or a 1, and integers, which can optionally be annotated with the possible bounds the integer could fall in. The type system enforces that integers can only contain values within their specified bounds. We currently do not leverage these bounds in our analysis, but we describe an approach to incorporate it in Section 7.1.

Container types are collections of data. These include vectors, tuples, lists, and records.

Registers and Memory. In the Sail language, registers can have any underlying type, but in practice, they are implemented as bit vectors. There is no representation of other resources like main memory within Sail specifications. Every architecture defines a memory read and write API by declaring extern
functions with the appropriate side effects. Since memory semantics are outside of the scope of the ISA, this functionality is undefined within Sail.

**ISA Specifications.** Currently, there are six ISAs defined in Sail: ARMv8, CHERI, MIPS, IBM Power, RISC-V, and x86 (see the Sail Manual[14] for a description of how these specifications were created). Although these ISAs lack complete definitions, the Sail project is open-source and can be improved by anyone in the research community.

### 4.2 Taint Analysis

Our analysis’s key idea is to use a static analysis technique called taint analysis to automatically identify side channels in an ISA specification. Taint analysis is a form of data-flow analysis to determine the set of program variables whose values are affected by the input data. In this analysis, variables are tagged with a label of either tainted or clean. A tainted tag indicates that the variable’s value is derived from the input data. Conversely, a clean tag denotes that the variable’s value is independent of the input. Taint analysis problems are typically defined in terms of sources, sinks, and propagation rules. Sources are producers of tainted values. Sinks are sensitive operations which should not be affected by tainted values. Propagation rules describe how different programming language constructs modify the taint tags of the variables involved. For instance, in the expression $C := A + B$, $C$ are marked as tainted if either of the variables $A$ or $B$ are tainted. If $A$ and $B$ are both clean, then $C$ must also be independent of the input, so the analysis would tag it as clean.

**Taint Analysis Use Cases.** Taint analysis is applicable in a variety of application domains. One use case of taint analysis is to determine if a SQL query uses untrusted user input in a web server setting. A malicious input string used in a SQL query could instruct the database to perform unwanted actions such as revealing extraneous information or even deleting data. In this problem, we define the source as the web request containing the string the user wants to look up in the database. The sink is the database call, and the propagation rules define a trusted SQL query sanitization function as an operation which converts a tainted string to a clean one. With this formulation, taint analysis can be used to determine if any code paths could pass an unsanitized string to the database.

**Properties of Taint Tracking.** There are several different properties a data-flow analysis could have. A flow-sensitive analysis considers each statement in the order they appear in the program. Path-sensitivity considers information implied by the predicates in conditional branches. For instance, in an if statement where the condition is $x=0$, the analysis will follow the consequent path, knowing statically that $x=0$. On the alternative path, the analysis will proceed...
with the knowledge that \(x \neq 0\). On the other hand, context-sensitive analysis follows function calls, keeping track of the calling function’s analysis context.

*Soundness* and *completeness* are two terms commonly used in the program analysis field to characterize the accuracy of an analysis. A sound analysis will never report a false negative. In the context of taint analysis, this means it will report every possible taint sink which could accept a tainted variable as input. However, this analysis may report false positives and include sinks which are never affected by the taint source at runtime. On the other hand, a complete analysis will never report a false positive. In other words, it will only report problematic taint sinks if it can definitively prove that the sink can be affected by the taint source. An ideal static analysis is both sound and complete, but this is typically infeasible.

5 Creating an ISA Side Channel Model

We extend the Sail compiler to apply static analysis to each of the ISA definitions. The compiler outputs a report of potential side channel leaks found in the ISA.

5.1 Taint Analysis Problem Formulation

Our work detects side channels in the ISA by applying taint analysis on the Sail specification. Because our goal is to prevent adversaries from learning secret values in the registers and in memory, we define reads from memory and general purpose registers (GPRs) as taint sources. In our threat model, attackers can measure the number memory accesses and faults in the victim program. We define the taint sinks as control flow statements which exhibit varying amounts memory accesses or faults between its branches. In the Sail language, the control flow expressions are if statements, switch case statements, and for loops.

5.2 Taint Propagation Algorithm

To traverse the AST, we begin by locating relevant top-level definitions. Function definitions are stored in a table so we can look up their AST nodes when following function calls for our context-sensitive analysis. Registers and global variables are tagged with their appropriate taints. For registers, we specify the GPRs as tainted, which we identify by consulting each architecture’s manuals[18, 19, 20, 21, 22, 23]. Instructions are implemented in special functions named `execute`, which are the entry points of our analysis. These function AST nodes are placed in a separate queue for processing. We then iterate through and apply our taint propagation rules in a new context for each of the entry points. Figure 2 describes the pseudocode outline of our algorithm.

Whenever we evaluate an AST node, we keep track of the taints of several variables in the evaluation context. Whenever an assignment occurs, the taints of the assigned variables are stored in this context. This implements a local
for each definition ∈ definitions do
  if definition.type == REGISTER then
    if is_general-purpose-register(definition.id) then
      mark(definition.id, TAINTED)
    else
      mark(definition.id, CLEAN)
  end if
  else if definition.type == GLOBAL VARIABLE then
    mark(definition.id, analyze(definition.ast))
  else if definition.type == FUNCTION then
    if definition.id == "execute" then
      entry_points.append(definition)
    else
      functions(definition.id) ← definition
    end if
  end if
end for
for each entry_point ∈ entry_points do
  analyze(entry_point.ast)
end for

Figure 2: Pseudocode to begin the analysis algorithm.

scope that can be cleared when a function call returns. This local scope can also be temporarily replaced with a fresh context when following a function call. Whenever an expression is evaluated, its return value is stored in the context as well. At the end of the evaluation function, we return a copy of the evaluation context containing only the variables which were affected by the AST node.

The table in figure 3 enumerates some of the taint propagation rules we use for the various Sail language constructs. We use several functions in our description. The GET_TAG function retrieves an identifier’s taint tag. The MERGE operation returns TAINTED if any of the argument expressions are tainted and CLEAN otherwise. The IS_CLEAN function evaluates an expression and returns true if the result is clean. The Sail Grammar column uses notation from the Sail Manual[14].

5.2.1 Blocks

Blocks are a set of sequentially executed statements. We evaluate every expression, which each return the variables which were changed as a result of executing that expression. We then use these changes to update the evaluation context for the next instruction and the final set of changed variables as a result of this block.
<table>
<thead>
<tr>
<th>AST Node</th>
<th>Sail Grammar</th>
<th>Returned Taint(s)</th>
<th>Other Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>{exp_1; \ldots; exp_n}</td>
<td>analyze(exp_n)</td>
<td>See Section 5.2.1.</td>
</tr>
<tr>
<td>Identifier</td>
<td>id</td>
<td>GET_TAG(id)</td>
<td></td>
</tr>
<tr>
<td>Literal</td>
<td>lit</td>
<td>CLEAN</td>
<td></td>
</tr>
<tr>
<td>Cast</td>
<td>(typ)exp</td>
<td>analyze(exp)</td>
<td></td>
</tr>
<tr>
<td>Function Call</td>
<td>id(exp_1, \ldots, exp_n)</td>
<td>analyze(functions(id).ast)</td>
<td>See Section 5.2.2</td>
</tr>
<tr>
<td>Infix Function</td>
<td>exp_1 id exp_2</td>
<td>MERGE(exp_1, exp_2)</td>
<td>See Section 5.2.2</td>
</tr>
<tr>
<td>Tuple</td>
<td>(exp_1, \ldots, exp_n)</td>
<td>(analyze(exp_1), \ldots, analyze(exp_n))</td>
<td></td>
</tr>
<tr>
<td>If Statement</td>
<td>if exp_1 then exp_2</td>
<td>if IS_CLEAN(exp_1) then MERGE(exp_2, exp_3) else TAINTED</td>
<td>See Section 5.2.3.</td>
</tr>
<tr>
<td></td>
<td>else exp_1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pattern Match</td>
<td>switch exp_0 { case pat_1 \rightarrow exp_1 ; case pat_n \rightarrow exp_n }</td>
<td>if IS_CLEAN(exp_0) then MERGE(exp_1, \ldots, exp_n) then TAINTED</td>
<td>See Section 5.2.3</td>
</tr>
<tr>
<td>For Loop</td>
<td>foreach (id from exp_1 to exp_2) exp_3</td>
<td>if IS_CLEAN(MERGE(exp_1, exp_2)) then analyze(exp_3) else TAINTED</td>
<td>See Section 5.2.4</td>
</tr>
<tr>
<td>Vector</td>
<td>[exp_1, \ldots, exp_n]</td>
<td>MERGE(exp_1, \ldots, exp_n)</td>
<td></td>
</tr>
<tr>
<td>Vector Access</td>
<td>exp_1[exp_2]</td>
<td>MERGE(exp_1, exp_2)</td>
<td></td>
</tr>
<tr>
<td>Vector Concat</td>
<td>exp_1 \cdot exp_2</td>
<td>MERGE(exp_1, exp_2)</td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>id := exp</td>
<td>analyze(exp)</td>
<td>Marks id with the returned taint.</td>
</tr>
<tr>
<td>Exit</td>
<td>exit exp</td>
<td>CLEAN</td>
<td>Counts a fault side effect in the report</td>
</tr>
</tbody>
</table>

Figure 3: This table describes a subset of the taint propagation rules we use in our analysis.
5.2.2 Function Calls

Our analysis is context-sensitive, so we follow most function calls, tagging the arguments with the taint from the calling function’s context. Some functions cannot be analyzed, however. In particular, our analysis cannot handle recursive functions and extern functions. These functions require a human to manually inspect the function and inform the analysis about the possible taints of the return values and any potential side effects that occur.

In the six architectures currently available in Sail, we only located four recursive functions. Each of these functions were pure and were simple to analyze by hand. The extern functions typically defined behavior like memory operations which are out of scope for the ISA. To determine these functions’ effects for the analysis, we consulted the available ISA documentation and made inferences from the functions’ names and declared side effects.

Functions can also be called with an infix syntax. However, these functions are always predefined binary operators in Sail, so we implement their analysis by merging our analysis results for their operands.

5.2.3 Branching Statements

Sail’s branching statements are if statements and switch case statements. We apply our analysis to each of the possible branches, and record the taints of every variable which are possibly assigned. If one branch taints a variable, and the other branch clears the taint, we will conservatively mark that variable as tainted. To implement this, we apply MERGE to each of the variables in the branches. When the condition variable in the if statement is tainted, the value of every variable which is modified in either branch depends on the value of the taint. Thus, if we find that the condition variable is tainted, we then replace the taints of the merged variable sets with the TAINTED tag. Additionally, because branching statements with tainted conditions are taint sinks, we check the number of memory accesses and faults in each branch. If there is a discrepancy between branches, our analysis adds the taint sink to the final report.

5.2.4 Loops

To evaluate looping constructs, we use fixed-point analysis. In this technique, our analysis on the expressions in the loop are repeatedly executed until the taints of the variables the loop modifies stop changing. With this approach, the analysis does not consider the true number of times a loop is run. Thus, when evaluating the results of a loop, we are unaware if our evaluation analyzes the loop more times than the loop body will actually be run. At the end of the loop analysis, we mark every variable as tainted if they were ever marked as tainted in an execution of the loop body since the loop might have stopped at that point. Additionally, if the number of times the loop executes depends on a tainted value, an attacker may learn secret information about the taint sources.
Thus, if the loop body contains operations that an attacker can measure, we add this taint sink to the analysis report.

5.3 Properties of Our Analysis

Our analysis is evaluates expressions in blocks in the order they appear, so it is flow-sensitive. Since it follows function calls wherever it is possible, the analysis is context-sensitive. However, our analysis is not path-sensitive since we do not attempt to infer any restrictions on variables based on the code paths that could be taken to get there.

In this analysis, we prefer soundness over completeness to guarantee that any potential side channel is detected. Thus, we designed the taint propagation rules to be as conservative as possible. This conservatism is very apparent in expressions that operate on container data types like vectors and records. Our rules will taint an entire container type if an operation sets any component of the container to a tainted value. Additionally, any value which is retrieved from a tainted container is assumed to be tainted, even though it may not be actually affected by the taint sources.

6 Evaluation

We perform our analysis on the six architectures defined in the Sail project. Additionally, the Sail project defines two variations of CHERI, one using 256-bit capabilities and one using 128-bit compressed capabilities. We evaluate both variants separately.

6.1 Analysis Performance

Though the analysis is context-sensitive, it terminates quickly in the architectures we examined. We believe this is primarily a result of the small code sizes in the specifications. The execution time for an analysis on a specification grows exponentially with respect to the number of lines of code in a specification. Figure 4 shows this exponential growth, and Figure 5 shows the range of code sizes for each specification.

6.2 Verifying the Analysis Implementation

To check the implementation of our taint propagation rules for bugs, we created a test suite of 22 different Sail functions. We designed these functions to check for edge cases in our taint propagation rules. These test cases also serve to identify regressions which break the soundness of the analysis during development. Additionally, we hand-verified several of the discovered analysis results after running it on the real ISA specifications.
Figure 4: Average time over 10 runs to complete an analysis vs. number of lines of code in an ISA specification.

Figure 5: Number of lines of specification code per architecture.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>ARMv8</th>
<th>CHERI-128</th>
<th>CHERI-256</th>
<th>MIPS</th>
<th>IBM Power</th>
<th>RISC-V</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taint Sinks</td>
<td>39</td>
<td>54</td>
<td>54</td>
<td>20</td>
<td>0</td>
<td>10</td>
<td>36</td>
</tr>
</tbody>
</table>

Figure 6: Number of Taint Sinks Per ISA Before Analysis

6.3 Analysis Results

Our analysis finds a total of 213 taint sinks over all of the ISA specifications. Many of these results are false positives. Luckily, a majority of these taint sinks result from a limited set of subroutines which execute potentially leaky behavior, limiting our manual analysis to a much smaller subset of the discovered taint sinks. Figure 6 describes the number of taint sinks in each architecture.

Through tracing the taint manually from a subset of the 213 candidate taint sinks, we are able to identify potential side channels in each of the ISAs besides IBM Power. Although these instructions are correctly implemented according to their specification, the definitions of the instruction semantics themselves could be problematic from a security standpoint. In practice, for correctness purposes, standard compilers already account for many of the problematic instructions we find. However, we believe it is still important to consider the security implications of these instructions’ error cases.

The following subsections describe the potential side channels we find in each of the architectures we examine. We also highlight six ISA design decisions that could leak information along with the type of information they leak.

6.3.1 ARMv8

Store-Exclusive. In ARM, the Store-Exclusive instruction stores a value only if a lock was successfully acquired. In the ISA specification, the status of the lock is determined when the translation lookaside buffer (TLB) translates the a virtual address to physical. Though the TLB is unimplemented in the ISA specification, we are certain that the lock status is affected by the memory address, which is obtained from the register. An attacker monitoring memory
accesses will be able to determine whether the secret memory value from the register was locked.

**Address Alignment Faults.** Another area where ARM displays data-dependent behavior is in certain instructions which require the provided memory addresses to be aligned. When an unaligned address is used in these instructions, an alignment fault occurs. Thus, the attacker would observe a program termination, which would inform them of the alignment of the secret address in memory.

### 6.3.2 MIPS

**Arithmetic Overflow Faults.** MIPS uses several arithmetic instructions which cause a fault when the result overflows the size of the register. These are the ADD, ADDI, and SUB instructions and their doubleword variants. Since one or both of their operands come from registers, an attacker detecting a fault would be able to determine that the secret data in the registers contain values which would cause an overflow.

**Conditional Trap Instructions.** Another class of instructions that leaks information is the group of conditional trap instructions, which cause a fault if the comparison operation on the two register operands is true. This is a straightforward example of a side channel, since an attacker will learn that the comparison is satisfied when they detect a fault.

All MIPS load and store instructions will cause a fault when the provided memory address to load is misaligned, exposing the same side channel we describe in the previous section.

### 6.3.3 CHERI

**Capability Exceptions.** CHERI extends the MIPS ISA and thus inherits the side channels present in MIPS. Additionally, it adds capabilities to the ISA, introducing an access control primitive. CHERI introduces new general-purpose capability registers which store both user data and capability-related metadata. This metadata is used to indicate whether an action using this capability register is allowed. If a capability-aware instruction is attempted when the register it uses lacks the required privileges, the ISA raises an exception. An attacker measuring for exceptions could potentially determine the capability-related metadata in the capability registers as a result of capability-related instruction.

However, since the analysis does not know if the components within collection data types are definitively affected by the secret information, this could be a false positive. For instance, if we only consider user data as sensitive and disregard the capability metadata stored in the same registers as the user data, an analysis which is able to separate the metadata and the sensitive user data may determine that these exceptions do not leak information.
if \( \text{cond} \) then
\[
\begin{align*}
y & \leftarrow x \\
x & \leftarrow 0
\end{align*}
\]
else
\[
\begin{align*}
x & \leftarrow 0 \\
y & \leftarrow 1
\end{align*}
\]
end if

Figure 7: Even if \( \text{cond} \) is tainted, \( x \) will be 0 no matter which branch is taken, so it should not be affected by a taint source by the end of the if statement. The current taint propagation rules will mark \( x \) as tainted, which could increase the false positive rate.

We analyzed the variant of CHERI with 128-bit compressed capability registers and the variant with 256-bit registers, and found the same potential side channels in both.

6.3.4 RISC-V
In RISC-V, memory addresses used in load and store operations must be aligned. This causes the same side channels as described in Section 6.3.1.

6.3.5 x86
Division by Zero Fault. In the division instruction, a fault occurs when there is division by zero. If the attacker notices that a division by zero fault occurs, they learn that the divisor is zero, which could have been a secret value from the registers or from memory.

7 Future Work

7.1 Improving the Analysis
Our analysis uses conservative taint propagation rules to preserve soundness, but these rules lead to many potential false positives. Several techniques could be used to lower the false positive rate while preserving soundness.

One potential solution is to make the analysis path-sensitive by associating variables with a set of constraints on their possible values. These constraints can be generated by using Sail’s own bounded integer type annotation, branch conditions in if statements, loop bounds, and assert statements. Using symbolic execution, we can update the variables’ constraints as they are updated through the program. By limiting the set of possible values a variable could have, we can introduce new ways to clear the taint of variables. Figure 7 illustrates a toy example of a program which could benefit from this analysis.

Our analysis can further improve by adding reasoning about the components within Sail’s data structures. For instance, if the analysis can show that a
subvector will never contain tainted values, it does not need to taint the entire vector when only part of it is updated. This would be especially helpful in the analysis for CHERI, as mentioned in Section 6.3.3.

Another avenue of improvement is to improve the Sail specification for each of the architectures. In particular, none of the ISAs specified in Sail describe the semantics for vector and floating point instructions. Having a complete specification of all of the ISA’s semantics will give the analysis better coverage of the side channels in the ISA.

7.2 Demonstrating Real Side Channel Attacks

Our analysis shows a number of possible side channel leakages in various ISAs. Further research can be conducted to demonstrate the side channel on real hardware and determine the bandwidth of any information leakages. If we discover instructions which leak information in practice, we can extend this discovery to demonstrate attacks on real-world applications which use the unsafe instruction.

7.3 Extending our Methodology to the Microarchitecture

Although our work focuses on the ISA, similar principles are applicable to discovering side channels in the microarchitecture. By applying sound taint analysis techniques to specifications written in a formally defined language, we can automatically identify side channel flaws in hardware. This can reduce the human effort to conduct an extensive security review over the specification.

8 Conclusion

In this paper, we propose a methodology to locate potential side channel leaks in an ISA specification, and we present a system implementing these techniques to analyze an ISA specification language for side channels. We apply this system to six different ISA specifications and show that the analysis is fast and discovers valid information leaks. Our approach produces a novel model of the ISA which can be utilized by side channel aware compilers to produce secure binaries which avoid unsafe machine instructions.

Acknowledgements. We thank Ashay Rane for his helpful guidance throughout the entire research process. We also thank Varun Adiga for his assistance in designing the analysis.

References


