Testing Software and Hardware against Speculation Contracts

Boris Köpf
Microsoft Research

NUS CS Research Week
07/01/22
Based on Joint Work With

• **Marco Guarnieri** (IMDEA Software Institute)
• Jose Morales (IMDEA Software Institute)
• **Oleksii Oleksenko** (Microsoft Research)
• Jan Reineke (Saarland University)
• Andres Sanchez (EPFL)
• Mark Silberstein (Technion)
• Pepe Vila (Arm)

... and lots of discussions with & feedback from colleagues at Microsoft
Performance is Fundamental

CPUs
Compilers
Virtual machines
Networks

minimize

time
space
energy

... consumption

...
Performance-enhancing techniques...

- Caching
- Concurrency
- Deduplication
- Compression
- ...
- ...
... and their Impact on Security

- Caching
- Concurrency
- Deduplication
- Compression
- ...

Reduce resource consumption on average

Exploit variations in resource consumption

Cache-timing attacks on AES

Loophole: Timing Attacks on Shared Event Loops in Chrome

Memory Deduplication as an Advanced Exploitation Vector

Spot me if you can:
Uncovering spoken phrases in encrypted VoIP conversations
Example: Cache Side-Channel

Main Memory

Cache

CPU

\[ b \leftarrow A[i]; \]
\[ c \leftarrow A[j]; \]

2005: First attacks on AES (Bernstein/Shamir et al.)
2014-...: Highly effective attacks using shared caches (Yarom et al/...)

\[ x_0 \leftarrow \text{FTD}[\text{Y0}] \]
\[ x_1 \leftarrow \text{FTD}[\text{Y1}] \]

...
... and their Impact on Security

- Caching
- Concurrency
- Deduplication
- Compression
- Speculative execution
Example: Speculative Leak

```c
if (x < A_size)
    y = A[x]
```

Wrong prediction? Roll back changes!
- Logical state
- Microarchitectural state

Branch Predictor
Spectre V1

\[ \text{void } f(\text{int } x) \]
\[ \text{if } (x < A\_size) \]
\[ y = B[A[x]] \]

\text{A\_size} = 16 \ldots \text{but what is stored in } A[128]?

1) Training
void f(int x)
    if (x < A_size)
        y = B[A[x]]
Covert Channels vs Side-channels

- Covert channels: Adversary = Sender & Receiver
- Side-channels: Adversary = Eavesdropper
Countermeasures

• Software-based countermeasures
  • Insertion of speculation barriers, speculative load hardening, ...
  • Rely on (often implicit) assumptions about underlying hardware

• Hardware-based countermeasures
  • InvisiSpec (Micro 18), NDA (Micro 19), STT (Micro 19), SPT (Micro 21), ...
  • Rely on (often implicit) assumptions about software

```java
if (i < A_size) {
    x = A[i];
    y = B[x*512];
}
```

- Delay loads until they retire
- Taint speculatively loaded data; delay loads to tainted addresses
- ... until they cannot be rolled back
This Talk: Co-design for Secure Speculation

1. Checking software for contract compliance

2. Hardware-software contracts for secure speculation

3. Checking CPUs for contract compliance
Speculation
Contracts
Speculation Contracts in a Nutshell

• Baseline contract: “constant-time programming”:
  • Obligation on software: Make sure secrets don’t affect loads, stores, branch targets
  • (Often implicit) obligation on hardware: Nothing except addresses of loads, stores, branch targets leaks
    • Technically: Make sure that all executions of a program that agree on addresses of loads, stores, and branch targets also agree on what a muarch attacker can observe
  • Note that both obligations are instance of non-interference (NI):
    \[ P \text{ satisfies NI } \iff \text{ for all } h, h', l: P(h, l) = P(h', l) \]

• Core idea: We generalize from executions and observations to capture the security properties of a wide range of speculation mechanisms
Examples of Contracts

• CT-Seq:
  • Observations: addresses of loads, stores, branch targets
  • Executions: sequential in-order

• CT-Spec:
  • Observations: addresses of loads, stores, branch targets
  • Executions: sequential in-order + “mispredicted” branches up to a bound

• Arch-Seq
  • Observations: addresses of loads, stores, branch targets + data that is loaded
  • Executions: sequential in-order

• CT-Bpas, CT-Spec-Bpas,...
What *is* a contract?

- A contract is a **labelled ISA semantics**, where labels correspond to the information that programs are allowed to leak during execution.

- **ISA:**

  Syntax
  \[
  \begin{align*}
  (\text{Expressions}) & \quad e & := & \quad n \mid x \mid \Theta e \mid e_1 \otimes e_2 \mid \text{ite}(e_1, e_2, e_3) \\
  (\text{Instructions}) & \quad i & := & \quad \text{skip} \mid x \leftarrow e \mid \text{load}\ x, e \mid \text{store}\ x, e \\
  & & & \mid \text{jmp}\ e \mid \text{beqz}\ x, \ell \mid \text{sbarr} \\
  (\text{Programs}) & \quad p & := & \quad i \mid p_1; p_2
  \end{align*}
  \]

- Core rules for CT-SEQ

\[
\begin{align*}
\text{LOAD} & \quad p(a(pc)) = \text{load}\ x, e \quad x \neq pc \\
& \quad n = \llbracket e \rrbracket(a) \\
& \quad \langle m, a \rangle \xrightarrow{\text{load}\ n} \langle m, a[pc \mapsto a(pc) + 1, x \mapsto m(n)] \rangle
\end{align*}
\]

\[
\begin{align*}
\text{BEQZ-SAT} & \quad p(a(pc)) = \text{beqz}\ x, \ell \\
& \quad a(x) = 0 \\
& \quad \langle m, a \rangle \xrightarrow{pc \ell} \langle m, a[pc \mapsto \ell] \rangle
\end{align*}
\]

\[
\llbracket P \rrbracket_{ct}^{\text{sed}}(\sigma) = \text{trace of observations}
\]
Core Rules for CT-Spec

**Step**

\[
p(\sigma(pc)) \neq \text{beqz } x, \ell \quad \sigma \xrightarrow{\text{seq}}_{ct}^{\text{spec}} \sigma' \quad \frac{\langle \sigma, \omega + 1 \rangle \cdot s \cdot \tau_{ct}^{\text{spec}}}{\langle \sigma', \omega \rangle \cdot s}
\]

**Rollback**

\[
s = \langle \sigma', \omega' \rangle \cdot s' \quad \frac{\langle \sigma, 0 \rangle \cdot s \cdot \text{pc} \cdot \sigma'(pc)^{\text{spec}}_{ct}}{\langle \sigma', 0 \rangle \cdot s}
\]

**Barrier**

\[
p(\sigma(pc)) = \text{spbarr} \quad \sigma \xrightarrow{\text{seq}}_{ct}^{\text{spec}} \sigma' \quad \frac{\langle \sigma, \omega + 1 \rangle \cdot s \cdot \tau_{ct}^{\text{spec}}}{\langle \sigma', 0 \rangle \cdot s}
\]

**Branch**

\[
p(\sigma(pc)) = \text{beqz } x, \ell \quad \ell_{\text{correct}} = \begin{cases} \ell & \text{if } \sigma(x) = 0 \\ \sigma(pc) + 1 & \text{otherwise} \end{cases}
\]

\[
\ell_{\text{mispred}} \in \{ \ell, \sigma(pc) + 1 \} \setminus \ell_{\text{correct}} \quad \omega_{\text{mispred}} = \begin{cases} w & \text{if } \omega = \infty \\ \omega & \text{otherwise} \end{cases}
\]

\[
\langle \sigma, \omega + 1 \rangle \cdot s \cdot \text{pc} \cdot \ell_{\text{mispred}}^{\text{spec}}_{ct} \quad \langle \sigma[pc \mapsto \ell_{\text{mispred}}], \omega_{\text{mispred}} \rangle \cdot \langle \sigma[pc \mapsto \ell_{\text{correct}}], \omega \rangle \cdot s
\]
Contracts form a Lattice

\[
\begin{align*}
\mathbf{1} & \rightarrow \mathbf{\text{spec}}_\text{arch} \\
\mathbf{\text{seq}}_\text{arch} & \rightarrow \mathbf{\text{seq}}_\text{ct} \\
\mathbf{\text{seq}}_\text{ct} & \rightarrow \mathbf{\text{seq-spec}}_\text{ct-pc} \\
\mathbf{\text{spec}}_\text{ct-pc} & \rightarrow \mathbf{\text{spec}}_\text{ct} \\
\mathbf{\text{spec}}_\text{ct} & \rightarrow \mathbf{1}_\top
\end{align*}
\]

\([\cdot]_2 \rightarrow [\cdot]_1\) means \([\cdot]_2\) leaks more information than \([\cdot]_1\)
Checking Programs for Contract Compliance
Checking Programs for Contract Compliance

• Contracts pose a verification condition on software:
  • Make sure secrets don’t affect contract traces
  • What is “secret” is defined by a policy $\pi$

  $\textbf{Definition 3}$ $(p \vdash NI(\pi, [\cdot ]))$. Program $p$ is non-interferent w.r.t. contract $[\cdot ]$ and policy $\pi$ if for all initial architectural states $\sigma, \sigma'$: $\sigma \sim_\pi \sigma' \Rightarrow [p](\sigma) = [p](\sigma')$.

• “Constant-time programming”: secret is part of architectural state
• “Sandboxing”: secret is memory that is not accessed during in-order execution

$$[p]_{\text{arch}}^{\text{seq}}(\sigma) = [p]_{\text{arch}}^{\text{seq}}(\sigma') \Rightarrow [p](\sigma) = [p](\sigma')$$
Tools for Checking Software

• Spectector
• Specfuzz
• Binsec/Haunted
• Pitchfork
• KleeSpectre
• SpecuSym
• ...

Spectector

1. Spectector symbolically executes a program wrt to a contract semantics to obtain pairs of (Path condition, Observation trace)

2. We query Z3 whether, for all $\sigma$, $\sigma'$ that satisfy the path condition, we have

$$\llbracket p \rrbracket_{arch}(\cdot) = \llbracket p \rrbracket_{arch}(\sigma') \Rightarrow \llbracket p \rrbracket_{ct}(\sigma) = \llbracket p \rrbracket_{ct}(\sigma')$$
Illustration: Kocher’s Examples

- **Ex 1: Vanilla Spectre 1**
- Ex 2: Move leak to local function
- Ex 3: Local function that can’t be inlined
- Ex 4: Left-shift y
- Ex 5: Use y as initial value of for loop
- Ex 6: Check bounds with mask rather than <
- Ex 7: Compare against last known-good value
- Ex 8: Use ?: operator
- Ex 9: Use separate value to communicate safety check
- Ex 10: Leak comparison result
- Ex 11: Use memcmp() to read memory for the leak
- Ex 12: Make index sum of two parameters
- Ex 13: Move safety check in inline function
- Ex 14: Invert lower bits of x
- Ex 15: Pass pointer to the length

```cpp
if (y < size) {
    temp &= B[A[y] * 512];
}
```
<table>
<thead>
<tr>
<th>Ex.</th>
<th>VISUAL C++</th>
<th>ICC</th>
<th>CLANG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UNP</td>
<td>FEN</td>
<td>UNP</td>
</tr>
<tr>
<td>01</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
<tr>
<td>02</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
<tr>
<td>03</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
<tr>
<td>04</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
<tr>
<td>05</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
<tr>
<td>06</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
<tr>
<td>07</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
<tr>
<td>08</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
<tr>
<td>09</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
<tr>
<td>10</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
<tr>
<td>11</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
<tr>
<td>12</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
<tr>
<td>13</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
<tr>
<td>14</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
<tr>
<td>15</td>
<td>🔄</td>
<td>🔄</td>
<td>🔄</td>
</tr>
</tbody>
</table>
Example 1

```c
if (y < size)
    temp &= B[A[y] * 512];
```

Clang V7.0.0
-O2
Speculative Load Hardening

```assembly
    mov    size, %rax
    mov    y, %rbx
    mov    $0, %rdx
    cmp    %rbx, %rax
    jbe    END
    cmovbe $-1, %rdx
    mov    A(%rbx), %rax
    shl    $9, %rax
    or     %rdx, %rax
    mov    B(%rax), %rax
    or     %rdx, %rax
    and    %rax, temp
```

%rax is -1 whenever y ≥ size. We can prove security.
Example 10

```
if (y < size)
    if (A[y] == k)
        temp &= B[0];
```

Clang V7.0.0
-O2
Speculative Load Hardening

We detect that A[0xFF..FF] can leak via control flow
Example 8

```c
temp &= B[A[y<size?(y+1):0]*512];
```

Intel ICC V19.0.0.117
-O2
w/ speculation barriers

```
  mov  y, %rdi
  lea  l(%rdi), %rdx
  mov  size, %rax
  xor  %rcx, %rcx
  cmp  %rax, %rdi
  cmovb %rdx, %rcx
  mov  temp, %r8b
  mov  A(%rcx), %rsi
  shl  $9, %rsi
  lfence
  and  B(%rsi), %r8b
  mov  %r8b, temp
```
Checking CPUs for Contract Compliance
Checking CPUs for Contract Compliance

• A CPU satisfies a contract if programs do not leak more information to a microarchitectural adversary than what the contract specifies
  • For all programs, whenever two executions agree on contract traces, they must also agree on hardware traces

• What is a “CPU”, what are “hardware traces”?
  1. “CPU” is an operational semantics with uarch components; hardware traces are obtained as a projection
     • Captures simple out-of-order CPU with 3-stage pipeline
     • Operates on registers, main memory, and reorder buffer
     • Stubs for caches, branch predictors, scheduler
  2. “CPU” is a fabricated chip; hardware traces are given by side-channel attack (e.g. Prime+Probe on L1D)
Contracts for Mechanisms for Secure Speculation

- STT protects data that is only transiently loaded
- STT still permits speculative leaks
Testing Black-box CPUs against Speculation Contracts

• Key observation: checking contract compliance can be done in a black-box fashion
  • For all programs, whenever two executions agree on contract traces, they must also agree on hardware traces

• Challenges:
  • How to cope with the intractable search space?
  • How to implement “contracts” for a realistic ISA?
  • How to obtain deterministic hardware traces?
• **Test case generator:** Creates DAG, adds terminators to blocks, populates with random instructions (from specified subsets) and operands (from specified subsets)

• **Input generator:** Generates random 32 bit numbers for registers, flags, and memory (1 or 2 pages)

• **Model:** Unicorn (QEMU-based), instrumented to collect traces + explore mispredicted branches

• **Executor:** Prime+Probe (on L1D) and Prime+Probe+assists (clear page table bit), based on nanoBench
  
  • Priming: Run each test case in a loop with different pseudorandom inputs to ensure muarch state is primed in a diverse but deterministic fashion
Results

<table>
<thead>
<tr>
<th></th>
<th>Target 1</th>
<th>Target 2</th>
<th>Target 3</th>
<th>Target 4</th>
<th>Target 5</th>
<th>Target 6</th>
<th>Target 7</th>
<th>Target 8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>Skylake</td>
<td>Skylake</td>
<td>Coffee Lake</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>V4 patch</strong></td>
<td>off</td>
<td>on</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instruction Set</strong></td>
<td>AR</td>
<td>AR+MEM</td>
<td>AR+MEM+VAR</td>
<td>AR+MEM+VAR</td>
<td>AR+MEM+CB</td>
<td>AR+MEM+CB+VAR</td>
<td>AR+MEM</td>
<td></td>
</tr>
<tr>
<td><strong>Executor Mode</strong></td>
<td>Prime+Probe</td>
<td>Prime+Probe+Assist</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Description of the experimental setups.

<table>
<thead>
<tr>
<th></th>
<th>Target 1</th>
<th>Target 2</th>
<th>Target 3</th>
<th>Target 4</th>
<th>Target 5</th>
<th>Target 6</th>
<th>Target 7</th>
<th>Target 8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CT-SEQ</strong></td>
<td>×</td>
<td>✓ (V4)</td>
<td>✓ (V4)</td>
<td>×</td>
<td>✓ (V1)</td>
<td>✓ (V1)</td>
<td>✓ (MDS)</td>
<td>✓ (LVI-Null)</td>
</tr>
<tr>
<td><strong>CT-BPAS</strong></td>
<td>×*</td>
<td>×</td>
<td>✓ (V4-var**)</td>
<td>×*</td>
<td>✓ (V1)</td>
<td>✓ (V1)</td>
<td>✓ (MDS)</td>
<td>✓ (LVI-Null)</td>
</tr>
<tr>
<td><strong>CT-COND</strong></td>
<td>×*</td>
<td>✓ (V4)</td>
<td>✓ (V4)</td>
<td>×*</td>
<td>×</td>
<td>✓ (V1-var**)</td>
<td>✓ (MDS)</td>
<td>✓ (LVI-Null)</td>
</tr>
<tr>
<td><strong>CT-COND-BPAS</strong></td>
<td>×*</td>
<td>×*</td>
<td>✓ (V4-var**)</td>
<td>×*</td>
<td>×*</td>
<td>✓ (V1-var**)</td>
<td>✓ (MDS)</td>
<td>✓ (LVI-Null)</td>
</tr>
</tbody>
</table>

* we did not repeat the experiment as a stronger contract was already satisfied. ** the violation represents a novel speculative vulnerability.

Table 3: Testing results. ✓ means Revizor detected a violation; × means Revizor detected no violations within 24h of testing. In parenthesis are Spectre-type vulnerabilities revealed by the detected violations.

- AR: in-register arithmetic, including logic and bitwise;
- MEM: memory operands and loads/stores;
- VAR: variable-latency operations (divisions);
- CB: conditional branches;
Detected Subtleties

• New variants of V1 & V4

```java
b = variable_latency(a)
if (...)  # misprediction
c = array[b]  # executed if the latency is short
```

• Speculative stores can modify the cache on Coffee Lake (but likely not on Skylake)
Detection Speed

• Time-to-violation

<table>
<thead>
<tr>
<th>Contract-permitted leakage</th>
<th>V4-type (Target 2)</th>
<th>V1-type (Target 5)</th>
<th>MDS-type (Target 7)</th>
<th>LVI-type (Target 8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>73'25'' (.7)</td>
<td>4'51'' (.9)</td>
<td>5'35'' (.7)</td>
<td>7'40'' (1.1)</td>
</tr>
<tr>
<td>V4</td>
<td>N/A</td>
<td>3'48'' (.7)</td>
<td>6'37'' (.8)</td>
<td>3'06'' (1.0)</td>
</tr>
<tr>
<td>V1</td>
<td>140'42'' (.6)</td>
<td>N/A</td>
<td>7'03'' (.8)</td>
<td>3'22'' (.3)</td>
</tr>
</tbody>
</table>
Summary

• We propose **HW/SW Contracts** as a framework for specifying security of speculative execution.
  • Captures unprotected speculation, InvisiSpec, Speculative Taint Tracking, ...
  • Can be used as a basis for secure programming

• We built **Spectector**, a tool to detect speculative leaks in software

• We built **Revizor**, a tool to test CPUs against contracts
  • Revizor generates random code snippets to find contract violations
  • Automatically surfaces V1, V4, LVI, MDS on x86 (Skylake and Coffee Lake)

• Many avenues for future work, including coverage, white-box analysis, and more expressive contracts
Links

• Hardware-Software Contracts for Secure Speculation - Microsoft Research (IEEE S&P ‘21)

• Spectector: Principled Detection of Speculative Information Flows - Microsoft Research (IEEE S&P ‘20)

• Revizor: Testing Black-box CPUs against Speculation Contracts (arxiv.org) (ASPLOS ‘22)

• Full Time Opportunities: Researcher (Side-channel Attacks and Defenses) in Cambridge | Research at Microsoft
Illustration: Kocher’s Examples

- **Ex 1: Vanilla Spectre 1**
- Ex 2: Move leak to local function
- Ex 3: Local function that can’t be inlined
- Ex 4: Left-shift y
- Ex 5: Use y as initial value of for loop
- Ex 6: Check bounds with mask rather than <
- Ex 7: Compare against last known-good value
- Ex 8: Use ?: operator
- Ex 9: Use separate value to communicate safety check
- Ex 10: Leak comparison result
- Ex 11: Use memcmp() to read memory for the leak
- Ex 12: Make index sum of two parameters
- Ex 13: Move safety check in inline function
- Ex 14: Invert lower bits of x
- Ex 15: Pass pointer to the length

```c
1 if (y < size)
2   temp &= B[A[y] * 512];
```
<table>
<thead>
<tr>
<th>Ex.</th>
<th><strong>VISUAL C++</strong></th>
<th><strong>ICC</strong></th>
<th><strong>CLANG</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UNP</td>
<td>FEN</td>
<td>UNP</td>
</tr>
<tr>
<td>01</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>02</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>03</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>04</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>05</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>06</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>07</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>08</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>09</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>10</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>11</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>12</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>13</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>14</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>15</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
</tbody>
</table>
Example 1

```c
if (y < size)
    temp &= B[A[y] * 512];
```

```assembly
mov    size, %rax
mov    y, %rbx
mov    $0, %rdx
cmp    %rbx, %rax
jbe    END
cmovbe $-1, %rdx
mov    A(%rbx), %rax
shl    $9, %rax
or     %rdx, %rax
mov    B(%rax), %rax
or     %rdx, %rax
and    %rax, temp
```

%rax is -1 whenever y ≥ size.
We can prove security
Example 10

```c
if (y < size)
    if (A[y] == k)
        temp &= B[0];
```

We detect that A[0xFF..FF] can leak via control flow

```assembly
mov    size, %rdx
mov    y, %rbx
mov    $0, %rax
cmp    %rbx, %rdx
jbe    END
cmovbe $-1, %rax
or     %rax, %rbx
mov    k, %rcx
cmp    %rcx, A(%rbx)
jne    END
cmovne $-1, %rax
mov    B, %rcx
and    %rcx, temp
jmp    END
```

Clang V7.0.0 -O2 Speculative Load Hardening
Example 8

```
temp &= B[A[y<size?(y+1):0]*512];
```

```
1 mov   y, %rdi
2 lea   l(%rdi), %rdx
3 mov   size, %rax
4 xor   %rcx, %rcx
5 cmp   %rax, %rdi
6 cmovb %rdx, %rcx
7 mov   temp, %r8b
8 mov   A(%rcx), %rsi
9 shr   $9, %rsi
10 lfence
11 and   B(%rsi), %r8b
12 mov   %r8b, temp
```
Core Idea 2: Checking Security

1. We **symbolically execute** speculative semantics to obtain pairs of (Path condition, Observation trace) $\langle \text{load}, \text{store}, \text{pc}, \text{start}, \text{rollback} \rangle$.

```
mov    size, %rax
mov    y, %rbx
mov    $0, %rdx
cmp    %rbx, %rax
jbe    END
cmovbe $-1, %rdx
mov    A(%rbx), %rax
shl    %9, %rax
or     %rdx, %rax
mov    B(%rax), %rax
or     %rdx, %rax
and    %rax, temp
```

\[
\begin{align*}
\text{start} \cdot \text{rollback} \cdot \tau & \quad \text{when} \quad y < \text{size} \\
\text{start} \cdot \tau \cdot \text{rollback} & \quad \text{when} \quad y \geq \text{size}
\end{align*}
\]

\[
\tau = \text{loadO}(A + y) \cdot \text{loadO}(B + (A[y] \times 512)||\text{mask})
\]

\[
\text{mask} = \text{ite}(y < \text{size}, 0x0, 0xFF..FF)
\]
Checking Contract Compliance vai

1. We **symbolically execute** program to obtain pairs of (Path condition, Observation Trace)

2. We **query Z3** whether, for all public $lo$ and all secret $hi, hi'$ that satisfy the path condition, we have
   - $\text{Obs}(lo,hi) = \text{Obs}(lo,hi')$ implies $\text{Obs}^{\text{Spec}}(lo,hi) = \text{Obs}^{\text{Spec}}(lo,hi')$

\[
\begin{align*}
\tau &= \text{loadO} \ (A + y) \cdot \text{loadO} \ (B + (A[y] \times 512) || \text{mask}) \\
\text{mask} &= \text{ite}(y < \text{size}, 0x0, 0xFF..FF)
\end{align*}
\]