# Opening Pandora's Box
## A Systematic Study of New Ways Microarchitecture Can Leak Private Data

Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel†, Adam Morrison‡, David Kohlbrenner§, Christopher W. Fletcher

University of Illinois at Urbana-Champaign, †Stanford University, ‡Tel Aviv University, §University of Washington

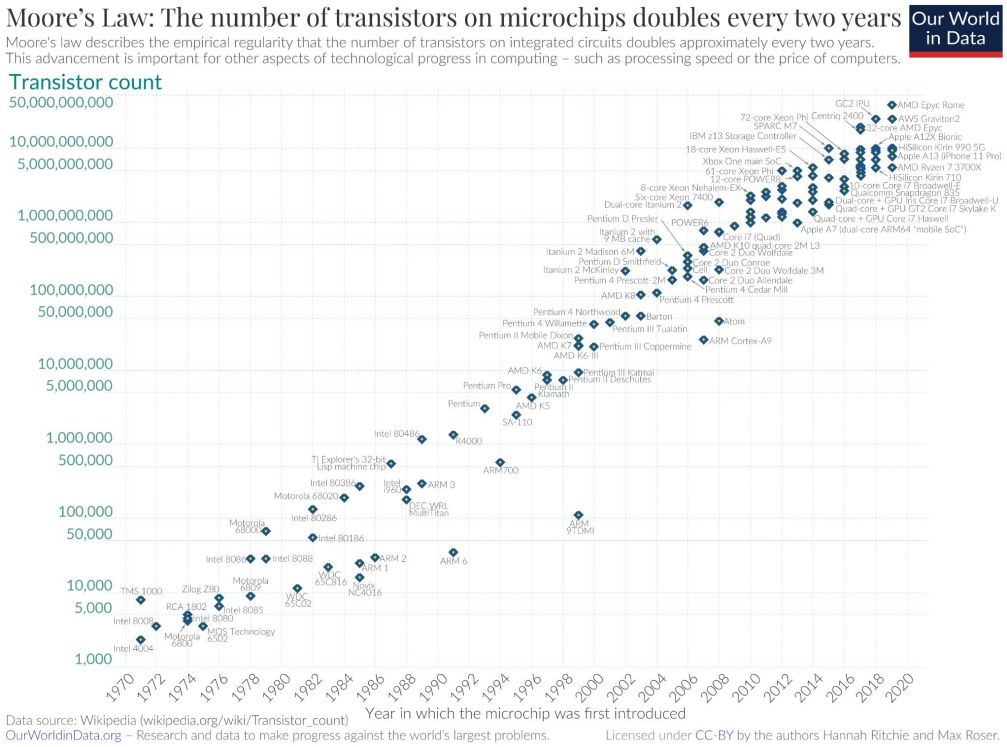
Presenters: Mindy Long, Joseph Zhang

# Questions to keep in mind

The authors continuously mention the slowing of Moore's law as one of the reasons for the increase in micro-architecture optimizations. **Should performance be of paramount concern** if each new optimization introduces new vulnerabilities in the system?

The paper says "An MLD for a given microarchitectural optimization is a stateless function that specifies 1) the inputs needed to describe the optimization’s functional behavior." **How are these inputs defined?** Doesn’t this depend on the specific implementation of the optimization? In question 1, for example, could we figure out what the input would be, or do we need more information?

# Overview




As Moore's law slows, microarchitects will introduce even **more processor optimizations**.

This leads to **more opportunities to take advantage of microarchitectural vulnerabilities** (think Spectre)

# Some previously un-analyzed microarchitectural optimizations

**Stateless Instruction-Centric**

1. Computation Simplification
2. Pipeline Compression

**Memory-Centric**

1. Register-file compression
2. Data memory-dependent prefetchers

**Stateful Instruction-Centric**

1. Silent Stores
2. Computation Reuse
3. Value Prediction

# Tools for analysis

## Microarchitectural leakage descriptors (MLDs)

- Simplified pseudocode to describe microarchitectural vulnerabilities

# Tools for analysis

## Microarchitectural leakage descriptors (MLDs)

- Simplified pseudocode to describe microarchitectural vulnerabilities

```
// Example 1: single-cycle addition.
mld single_cycle_alu(Inst i1): return 0
```

# Tools for analysis

## Microarchitectural leakage descriptors (MLDs)

- Simplified pseudocode to describe microarchitectural vulnerabilities

```
// Example 1: single-cycle addition.
mld single_cycle_alu(Inst i1): return 0
```

TABLE II: Optimization classification based on MLD signature.

| | Instruction (Inst) | | | Memory (Arch) - IV-D |
|---|---|---|---|---|
| | Stateless - IV-B | Stateful - IV-C | | |
| | | Uarch | Arch | |
| Comp. simplification | √ | | | |
| Pipeline compression | √ | | | |
| Silent stores | | | √ | |
| Computation reuse | | √ | | |
| Value prediction | | √ | | |
| Reg. file compression | | | | √ |
| D-memory prefetching | | | | √ |

# Memory-Centric Optimizations

# Why are memory optimizations particularly scary?

They leak data ***regardless*** of how it is computed.

This falls outside the model for writing constant-time programs and indiscriminately risks all registers and data memory.

# Register-File Compression

**Register-file compression** seeks to increase the number of available registers by checking if a register already stores a computation value.

**Implementation:** Check if result of instruction is already in a register file

a. Yes: return register originally supposed to hold the result to the “free pool”
b. No: Allocate register to store result

# Data Memory-Dependent Prefetchers

**Data memory-dependent prefetchers** take into account contents of data memory directly, rather than just addresses to data memory.

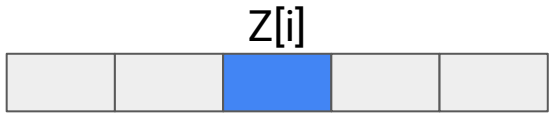
This most directly affects ***pointer chasing*** (applications in tensor algebra and graphs).

# Data Memory-Dependent Prefetchers

**Data memory-dependent prefetchers** take into account contents of data memory directly, rather than just addresses to data memory.

This most directly affects ***pointer chasing*** (applications in tensor algebra and graphs).

**Pointer Chasing: X[Y[Z[i]]]**

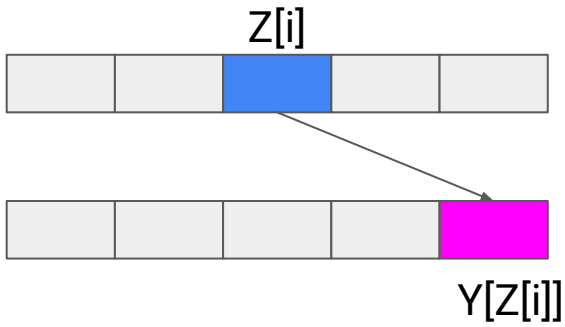


# Data Memory-Dependent Prefetchers

**Data memory-dependent prefetchers** take into account contents of data memory directly, rather than just addresses to data memory.

This most directly affects ***pointer chasing*** (applications in tensor algebra and graphs).

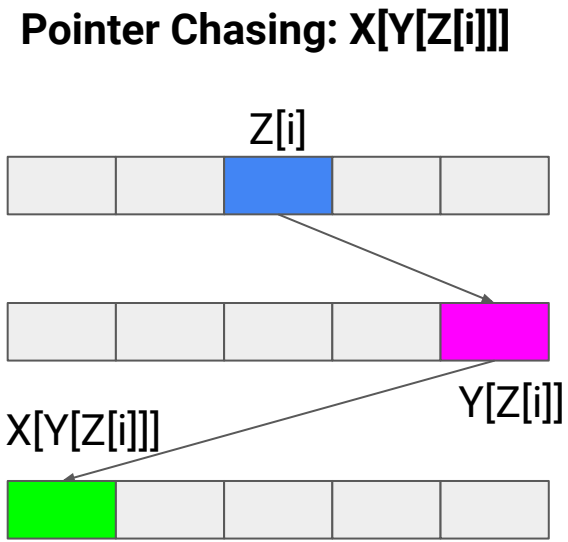
**Pointer Chasing: X[Y[Z[i]]]**



# Data Memory-Dependent Prefetchers

**Data memory-dependent prefetchers** take into account contents of data memory directly, rather than just addresses to data memory.

This most directly affects ***pointer chasing*** (applications in tensor algebra and graphs).

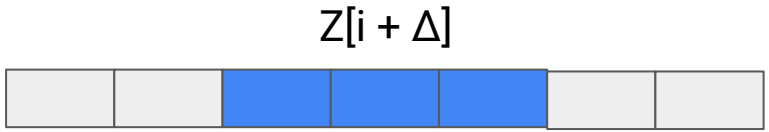


# Data Memory-Dependent Prefetchers

What if I want an entire row of a matrix?

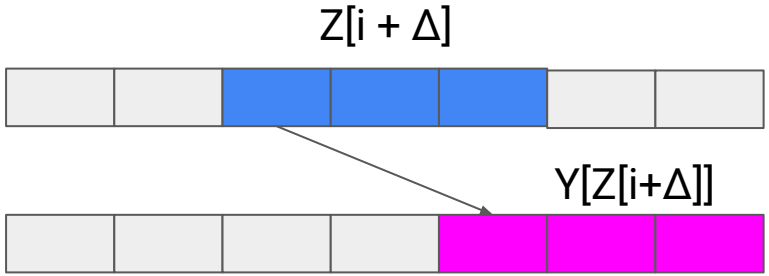
# Data Memory-Dependent Prefetchers

What if I want an entire row of a matrix?



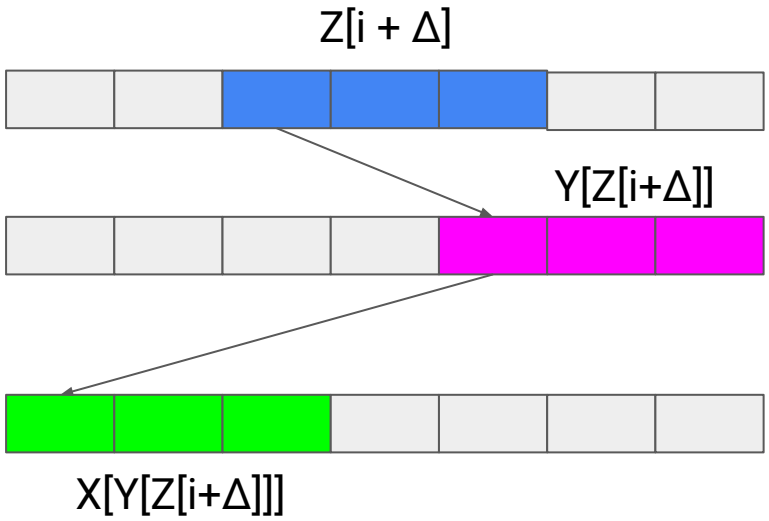
# Data Memory-Dependent Prefetchers

What if I want an entire row of a matrix?



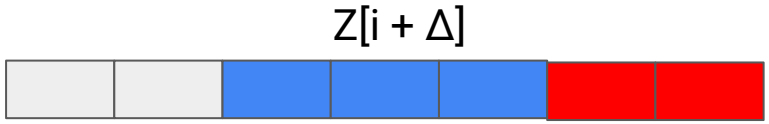
# Data Memory-Dependent Prefetchers

What if I want an entire row of a matrix?



# Data Memory-Dependent Prefetchers

What would happen if 😈 controlled some portion of memory outside of Z?



# Data Memory-Dependent Prefetching

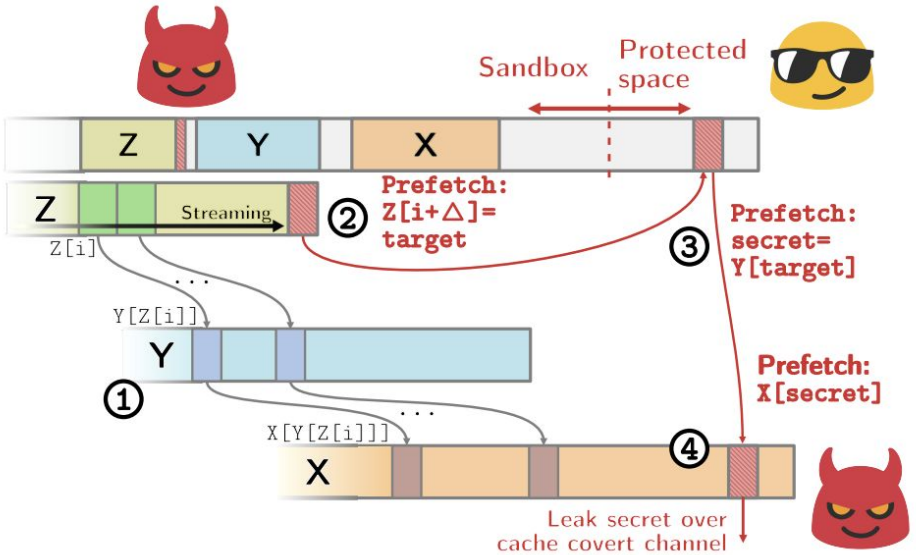



Fig. 1: Indirect-memory prefetcher leaking all of program memory (forming a *universal read gadget* [17]) in the sandbox setting. The attacker can choose which word of private data it wants to leak by setting the value target relative to the base address of array Y. The victim's sunglasses represent the sandbox's software-level memory-safety checks.

# Stateless Instruction-Centric

# Computation Simplification

**Computation simplifications** are techniques (e.g. skip zero-multiply, replace divide with a right shift by 2) to simplify or eliminate operations when operand values satisfy certain conditions

```
// Example 2: zero-skip multiply.
mld zero_skip_mul(Inst i1) : return ∨ᵢ i1.arg.vᵢ == 0
```

# Pipeline Compression

**Pipeline compression** compresses data as it moves through the processor pipeline.

**Principle:** Data is only as “wide” as its most significant non-zero bit

Implementations:

- packing multiple “narrow” operands to move between processor stages
- breaking data into words/bytes/bits before moving

# Pipeline Compression

**Pipeline compression** compresses data as it moves through the processor pipeline.

**Principle:** Data is only as “wide” as its most significant non-zero bit

Implementations:

- packing multiple “narrow” operands to move between processor stages
- breaking data into words/bytes/bits before moving

```
// Example 4 (Section IV-B): arithmetic unit operand
packing [24]. i1 and i2 must share the same execution unit
type.
mld operand_packing(Inst i1, Inst i2):
  return msb(i1.arg.v0) < 16 ∧ msb(i1.arg.v1) < 16 ∧
         msb(i2.arg.v0) < 16 ∧ msb(i2.arg.v1) < 16
```

# Security Concerns

**Threat to constant-time programming**

Even bitwise operations won’t be constant-time.

**Optimizations depend on multiple instructions.**

In pipeline compression, an attacker can choose its i2 instruction so packing optimization occurs as a function of the i1 instruction

# Proof of Concept – Data Memory-Dependent Prefetching

Attacker designed to trigger IMP

IMP prefetches “assuming” everything will be within array bounds, attacker may instead violate this assumption

- IMP detects indirect access patterns by monitoring Z[i]s and addresses for indirections Y[Z[i]]s
- It uses this information to solve for &X[0], &Y[0] to prefetch, assuming Z[i]s will be in bounds

Prefetch buffer – some prefetchers do not fill cache unless prefetched data read, not applied to every cache level

```
1  BPF_ARRAY(Z, int, N);
2  BPF_ARRAY(Y, int, N);
3  BPF_ARRAY(X, int, N);
4  int attacker () {
5    int j = 0; int *v;
6    for (j=0;j<N-1;j++) {
7      int i = j;
8      v = Z.lookup(&i);
9      if (!v) return 0;
10     v = Y.lookup(v);
11     if (!v) return 0;
12     v = X.lookup(v);
13     if (!v) return 0;
14     if (!*v) return 0; }
15   return 0; }
```

(a) Attacker eBPF source.

```
1  mov    0x0(%rsi),%eax # eax = Z[i]
2     # bounds check Z[i] < len(Y):
3  cmp    $0x40,%rax
4  jae    0x000000000000007f
5  shl    $0x3,%rax
6  add    %rdi,%rax # rax = &Y[Z[i]]
7  jmp    0x0000000000000081
8  xor    %eax,%eax
9  cmp    $0x0,%rax
10 je     0x00000000000000db
11 movabs $0x...,%rdi # rdi = X object
12 mov    %rax,%rsi
13 add    $0xd0,%rdi # rdi = &X array
14    # eax = Y[Z[i]]:
15 mov    0x0(%rsi),%eax
```

(b) Attacker eBPF JITed assembly snippet.

Fig. 7: Attacker program to break out of the eBPF sandbox using the 3-level indirect-memory prefetcher.

# Stateful-Instruction Optimizations

# Computation Reuse

Detect redundant computations in hardware (non-speculative)

1) Identify computation
2) Table lookup (e.g. by operand value/register)
3) Use result on hit (skip computation)

```
// Example 6 (Section IV-C): dynamic instruction reuse, Sv
variant [74]. reuse_buffer is the PC-indexed memoization
table that records each memoized instruction's operand
values.
mld instruction_reuse(Inst i1, Uarch reuse_buffer):
  return ∧i i1.arg.vi == reuse_buffer[i1.pc][i]
```

# Computation Reuse

Potentially leaked information depends on the choice of lookup key

- Values: may leak operand values (typically achieves higher reuse)
- Operand register IDs: only info about what instruction is executing

How would this change the MLD?

```
// Example 6 (Section IV-C): dynamic instruction reuse, Sv
variant [74]. reuse_buffer is the PC-indexed memoization
table that records each memoized instruction's operand
values.
mld instruction_reuse(Inst i1, Uarch reuse_buffer):
  return  ∧i  i1.arg.vi == reuse_buffer[i1.pc][i]
```

# Value Prediction

Increase instruction-level parallelism by breaking instruction dependencies (predicts the result of instructions before they are computed – speculative)

Nearly all proposals are threshold based (do not make predictions unless predictions are sufficiently high confidence)

```
// Example 7 (Section IV-C): value prediction [75].
prediction_table is the PC-indexed predictor table where
each entry contains a confidence conf (an unsigned
number) and a predicted value prediction.
mld v_prediction(Inst i1, Uarch prediction_table):
  return prediction_table[i1.pc].conf ||
         prediction_table[i1.pc].prediction == i1.dst.v
```

# Silent Stores

Skip “silent stores” that do not change contents of memory

Improves memory throughput by freeing up memory write port, but this resource usage difference is not immediately useful as a measurable timing difference

(Out-of-order pipelines are very good at preventing stalls from stores to memory)

```
// Example 5 (Section IV-C): silent stores [25]. i1 must be
a store.
mld silent_stores(Inst i1, Arch data_memory):
  return i1.data.v == data_memory[i1.addr.v]
```

# Silent Stores

Example: check for silent store at store retirement

- Inputs: in-flight store data, data stored at address
- Distinct behaviours: skip performing store, perform store as usual

In this case, attacker only needs to control one of the inputs to learn the other

```
// Example 5 (Section IV-C): silent stores [25]. i1 must be
a store.
mld silent_stores(Inst i1, Arch data_memory):
  return i1.data.v == data_memory[i1.addr.v]
```

# Proof of Concept – Silent Stores

**Read-port stealing scheme:**

Issue a Silent-Store-Load, as soon as the store address resolves and there is a free load port, that reads the contents of memory at the store address

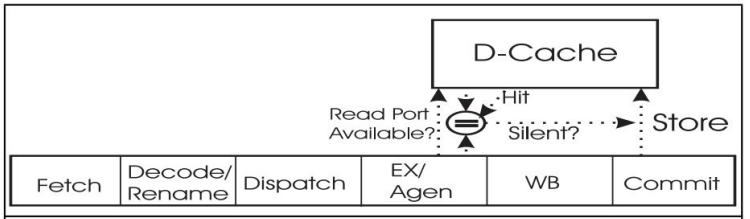



**FIGURE 3. Read port stealing performs a load and compare only if a cache port is idle.**

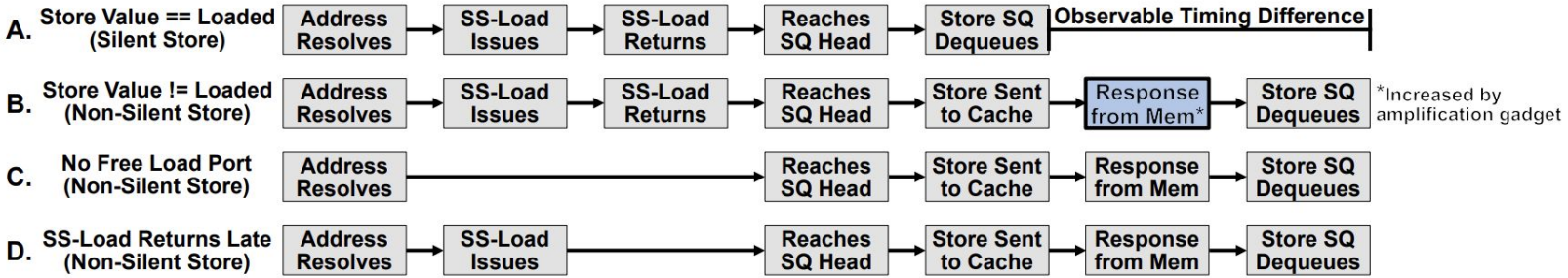



Fig. 4: The different possible sequences of actions taken by a store given the read-port stealing scheme from [86].

# Proof of Concept – Silent Stores (Amplification Gadget)

**Goal:** create a large timing difference depending on whether an attacker-chosen store is silent or not

**Example implementation:** Here, the gadget is part of the victim program

```
// Preconditions:
// - A, S have resolved
// - line(A) not present in cache
// - line(S) present in cache
// - set(S) != set(A), set(S) == set(A')
load A' <- (A)  // delay sub-gadget
load _  <- (A') // flush sub-gadget
store D -> (S)  // target store
```

Fig. 5: A single-threaded (i.e., inline with the victim program) amplification gadget for a release-consistency memory model and a direct-mapped cache. `line(X)` refers to the cache line associated with address `X`. `set(X)` refers to the cache set occupied by `line(X)`.

# Proof of Concept – Silent Stores (Amplification Gadget)

**Example implementation:** Delay/flush sub-gadgets using cache contention

**Delay gadget** takes a long time to execute

**Flush gadget** depends on delay gadget (to execute after SS-load) and removes target line from cache

```
// Preconditions:
// - A, S have resolved
// - line(A) not present in cache
// - line(S) present in cache
// - set(S) != set(A), set(S) == set(A')
load A' <- (A)  // delay sub-gadget
load _  <- (A') // flush sub-gadget
store D -> (S)  // target store
```

Fig. 5: A single-threaded (i.e., inline with the victim program) amplification gadget for a release-consistency memory model and a direct-mapped cache. `line(X)` refers to the cache line associated with address `X`. `set(X)` refers to the cache set occupied by `line(X)`.

# Proof of Concept – Silent Stores (Amplification Gadget)

**Step 1**

Delay gadget makes sure **SS-load completes before target store performed**

SS-load done concurrently, returns first, so silent store candidacy checked before target store performed

```
// Preconditions:
// - A, S have resolved
// - line(A) not present in cache
// - line(S) present in cache
// - set(S) != set(A), set(S) == set(A')
load A' <- (A)  // delay sub-gadget
load _  <- (A') // flush sub-gadget
store D -> (S)  // target store
```

Fig. 5: A single-threaded (i.e., inline with the victim program) amplification gadget for a release-consistency memory model and a direct-mapped cache. `line(X)` refers to the cache line associated with address `X`. `set(X)` refers to the cache set occupied by `line(X)`.

# Proof of Concept – Silent Stores (Amplification Gadget)

**Step 2**

Delay sub-gadget returns after SS-load completes, the flush sub-gadget may now execute

Target line removed from cache **after SS-load completes, before target store performed**

```
// Preconditions:
// - A, S have resolved
// - line(A) not present in cache
// - line(S) present in cache
// - set(S) != set(A), set(S) == set(A')
load A' <- (A)  // delay sub-gadget
load _  <- (A') // flush sub-gadget
store D -> (S)  // target store
```

Fig. 5: A single-threaded (i.e., inline with the victim program) amplification gadget for a release-consistency memory model and a direct-mapped cache. `line(X)` refers to the cache line associated with address `X`. `set(X)` refers to the cache set occupied by `line(X)`.

# Proof of Concept – Silent Stores (Amplification Gadget)

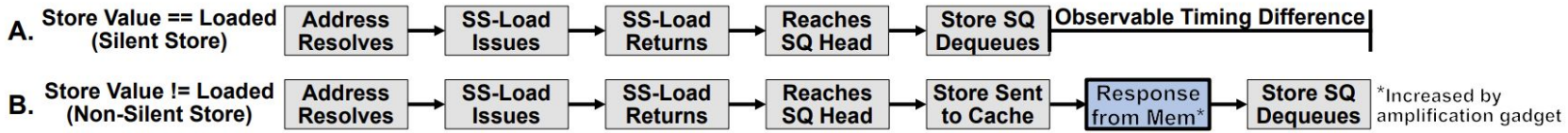
**Step 3**

For non-silent store, dequeue happens **only when target line filled in cache**, so timing difference between case A and B is amplified!

Creates a delay proportional to cache miss latency

```
// Preconditions:
// - A, S have resolved
// - line(A) not present in cache
// - line(S) present in cache
// - set(S) != set(A), set(S) == set(A')
load A' <- (A)  // delay sub-gadget
load _  <- (A') // flush sub-gadget
store D -> (S)  // target store
```

Fig. 5: A single-threaded (i.e., inline with the victim program) amplification gadget for a release-consistency memory model and a direct-mapped cache. `line(X)` refers to the cache line associated with address `X`. `set(X)` refers to the cache set occupied by `line(X)`.



# Proof of Concept – Silent Stores (Bitslice AES128)

Cloud threat model:

Attacker and victim trigger encryption calls with known plaintexts in worker thread

Victim is repeatedly encrypting the same public data (e.g., a packet header)

Encryption worker leaves temporary variables on the stack

It turns out there are eight locations storing these intermediate values that are enough to reconstruct victim key

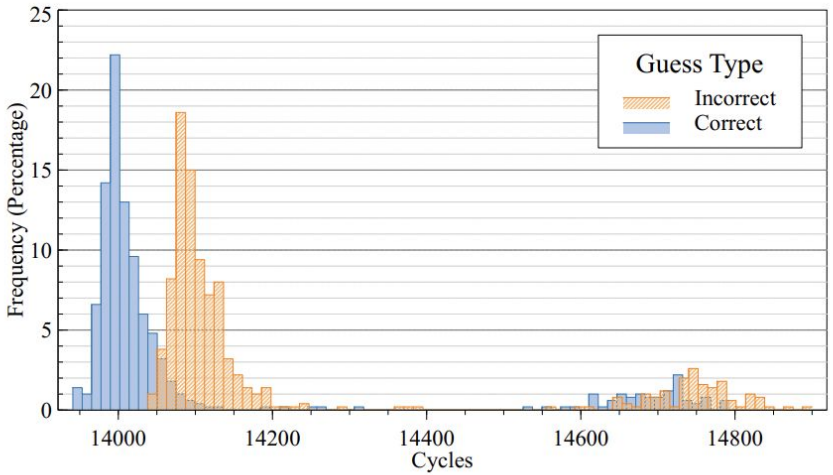



Fig. 6: Histogram of runtimes for BSAES when the amplification gadget is applied to one of the eight stores that overwrites AES state. That is, timing differences reflect whether a single dynamic store instruction was silent or not.

# Proof of Concept – Silent Stores (Bitslice AES128)

Victim encrypts data, leaving behind **intermediate values** on the stack

Attacker encrypts data, to see if the writes to the intermediate value locations **trigger silent stores**

Attacker knows its own key and can modify its own plaintext, so it can try triggering silent stores with different values until one matches

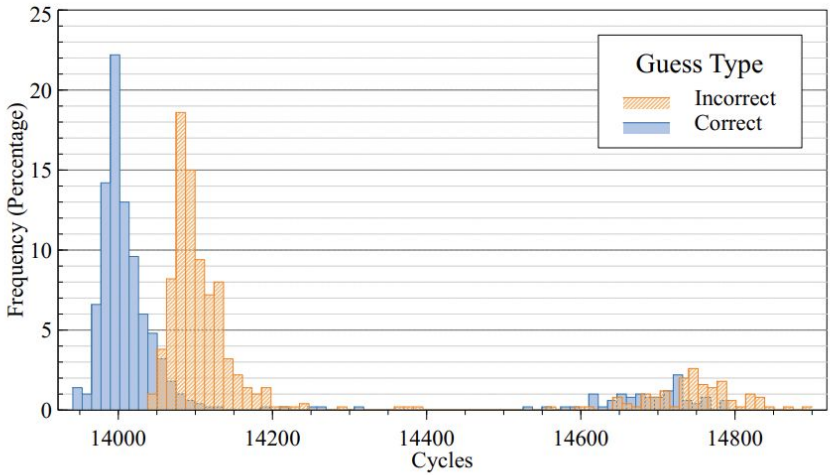



Fig. 6: Histogram of runtimes for BSAES when the amplification gadget is applied to one of the eight stores that overwrites AES state. That is, timing differences reflect whether a single dynamic store instruction was silent or not.

# Things to consider going forward

## Possible defense strategies

- Attempts to block all microarchitectural attacks
- Retrofitting constant-time programming
- Architecting security-conscious microarchitecture

## Additional optimizations with novel security implications

- The analysis in this paper may be incomplete

## Processor attack landscape going forward

- Power/energy microarchitecture-related attacks

# Discussion Questions

The authors continuously mention the slowing of Moore's law as one of the reasons for the increase in microarchitecture optimizations.

**Should performance be of paramount concern** if each new optimization introduces new vulnerabilities in the system?

# Discussion Questions

"An MLD for a given microarchitectural optimization is a stateless function that specifies the inputs needed to describe the optimization's functional behavior."

- **How are these inputs defined?**
- Doesn't this depend on the specific implementation of the optimization?
- In question 1, for example, could we figure out what the input would be, or do we need more information?

(From question 1: In conventional caches, the size of a physical entry in the data array generally matches the size of the cache line. In compressed caches, however, a single data entry can contain the data of multiple lines.)

# Discussion Questions

From some preliminary searching, Gem5 seems to be a software based emulator.

- How valid are software based emulation strategies for computer architecture for determining the scope and severity of uarch attacks?
- Are there major differences that must be taken into account when going to real hardware?