μSTT: Microarchitecture Design for Speculative Taint Tracking

Boru Chen[†], Rutvik Choudhary[§], Kaustubh Khulbe[§], Archie Lee[†], Adam Morrison[*], Christopher W. Fletcher[†]
[†]University of California, Berkeley, CA, USA, {boruchen,seungwoo415,cwfletcher}@berkeley.edu
[§]University of Illinois Urbana-Champaign, IL, USA, {rutvikc2,kkhulbe2}@illinois.edu
[*]Tel Aviv University, Israel, mad@cs.tau.ac.il

***Abstract*—Speculative execution attacks exploit malicious speculation to leak sensitive data via microarchitectural covert channels. Speculative Taint Tracking (STT) is a state-of-the-art hardware mechanism that blocks such threats by tainting data flowing from speculative loads, untainting data once all its dependencies are not speculative, and delaying instructions that create covert channels until their inputs are untainted. However, STT's hardware feasibility remains unclear due to a lack of detailed hardware cost analysis.**

This paper presents the first in-depth hardware cost analysis of STT and identifies two key challenges: (1) the logic delay of taint propagation, which grows with rename width, and (2) area overhead from instruction delaying, which requires expensive CAM-style logic to enforce speculation safety.

To address these, we propose a new microarchitecture for STT, called μSTT. μSTT is based on two new mechanisms. First, the *Age Matrix* is a shallow taint propagation circuit that removes 85% of the logic delay overhead of prior STT designs, while only adding 36% more area at the default rename width of 8. Second, the *impede micro-op* implements instruction delaying in a fashion that increases STT's performance overhead by only 5 percentage points (from 16% to 21%), while replacing bespoke STT hardware with existing RAW dependency tracking. Together, these contributions reduce STT's hardware complexity and cost in the context of high-end wide-issue processor designs.

***Index Terms*—Secure Speculation, Microarchitecture, Taint Tracking, Speculative Execution Attacks**

I. Introduction

Speculative execution attacks [16], [19]–[23] exploit control-flow (e.g., branch prediction [41]) and data-flow (e.g., value prediction [25]) speculation in modern processors to leak sensitive data from a victim's program memory via microarchitectural covert channels. These attacks intentionally trigger mispredictions, causing the processor to speculatively execute *transient instructions* (instructions that will eventually be squashed) that can transmit data through covert channels. For example, consider Spectre V1, given by `if (i < N) { arr2[arr1[i]] }`. Here, an attacker transmits (leaks) the contents of an arbitrary memory location (out of bounds of `arr1[i]`) through a cache-based covert channel (given by the lookup into `arr2`) by causing the branch predictor to mispredict and take the branch even when `i < N` resolves to false architecturally.

This paper considers the state-of-the-art hardware-based defense against speculative execution attacks called STT [45]. STT propagates *taint* to prevent speculative data flows from reaching covert channels. For example, it would delay the load into `arr2`. As soon as the branch resolves, STT dynamically *untaints* all flows that begin with a load in the shadow of the resolved branch. That is, it will allow the load into `arr2` to proceed as soon as the branch resolves.

STT is a compelling implementation candidate to mitigate speculative execution attacks for a number of reasons. For example, it is software-transparent, relatively low performance overhead and only requires hardware changes to the processor core (as opposed to requiring changes to the ISA or processor core + memory hierarchy [4], [42]).

Yet, challenges to adoption remain. The original STT paper lacks a detailed hardware cost analysis. We find this overhead to be substantial. In particular, STT's taint propagation logic is implemented in the rename stage and significantly increases the logic delay (to linear in the rename width). Simultaneously, STT's instruction delay/untainting logic requires extensive changes (in the form of state, broadcast buses and CAM-style logic) to the processor backend.

A. This Paper

This paper ameliorates these issues by proposing μSTT: a microarchitecture design for STT that features a shallow taint propagation circuit and a low-complexity hardware instruction delay mechanism. In the following, we refer to the original STT design as OG-STT.

Reducing Taint Propagation Delay. To support fast untainting, OG-STT must compute each renamed operand's *youngest root of taint* (yrot), which tracks the youngest load that register depends on. This results in a rename stage with circuit depth linear in the rename width: the yrot of the output register of an instruction is the younger of the yrots of its inputs and its inputs could come from the same rename group.

We address this by decoupling yrot computation into two parallel steps: (1) identifying the source yrots each instruction depends on, and (2) computing the younger-than relationships among all source yrots using what we call an *Age Matrix*. These two steps are later merged to assign each instruction the youngest yrot among its dependencies. We implement our design in Verilog and synthesize it using the open-source Skywater 130nm technology process [1]. At the default rename width of 8, our Age Matrix-based circuit removes 85% of the

This work was supported in part by NSF grants CNS-1954521, CNS-1942888, CNS-2154183, and also by a gift from Intel.

logic delay overhead of OG-STT, with only a 36% increase in area.

Area of Instruction Delaying/Untainting. STT requires delaying instructions that can create covert channels, until their yrots are older than the current *visibility point (VP)*, which marks the youngest non-speculative load. In OG-STT, this is implemented by associating a yrot field with each reservation station entry and performing expensive CAM-style VP comparisons. This approach incurs substantial area overhead.

We propose an alternative approach that reuses existing processor hardware. The idea is to delay instructions using a new micro-op, called the *impede* micro-op. In this design, yrots are stored using the physical register fields in the reservation station, and standard data dependency logic is reused to delay instruction issue until the yrot crosses the VP. We evaluate this scheme using the Gem5 simulator on SPEC CPU2017 benchmarks and find that it only marginally increases STT's performance overhead (from 16% to 21%), while significantly reducing hardware complexity.

Contributions. We make the following contributions:

1) We present the first in-depth analysis of STT and identify two key challenges that hinder its adoption in high-performance superscalar processors: critical path delay in taint propagation logic, and complexity/area overhead in instruction delaying/untainting.
2) We propose a shallow taint propagation circuit, called the Age Matrix scheme, that reduces logic depth.
3) We propose a low-complexity untaint propagation mechanism, called the impede micro-op scheme, which achieves instruction delaying/untainting while leveraging existing RAW dependency tracking.
4) We synthesize our Age Matrix-based taint propagation circuit. Our design removes 85% of the logic delay overhead of OG-STT with only a 36% increase in area at a rename width of 8.
5) We evaluate the impede micro-op scheme on SPEC CPU2017 benchmarks [2]. Our design only marginally increases STT's performance overhead (from 16% to 21%), while significantly reducing hardware complexity.

The evaluation infrastructure and code used in this paper have been open-sourced and are available at https://github.com/FPSG-UIUC/uSTT.

II. Threat Model & Scope

We wish to protect the confidentiality of a victim program's data through speculative execution. We adopt the same attacker model used by prior hardware defenses [44], [45]: (i) The attacker is allowed to monitor covert channels from anywhere that a thread can run in the processor (co-located with the victim on the same physical core, on a different physical core, etc.). (ii) The attacker is allowed to monitor any covert channel available to it (e.g., through the caches [26], [40], TLBs [13], [35], on-chip interconnects [12], [28], [37], arithmetic ports [5], [7], and BTBs [24], [43], [47]). (iii) The attacker is allowed to induce arbitrary mis-speculation in the victim program.

III. Background

A. Speculative Execution Attacks & Defenses

Speculative execution attacks exploit control- (e.g., branch prediction [41]) or data-flow (e.g., value prediction [25]) speculation in modern processors to leak sensitive data via microarchitectural covert channels. These attacks manipulate the processor into misprediction, causing it to speculatively execute transient instructions (instructions that will eventually be squashed) that can transmit data via covert channels. Researchers have uncovered numerous speculative execution attacks exploiting various forms of control-flow [21]–[23] and data-flow speculation [16], [19], [20].

Following most prior defenses [44], [45], our goal is to protect *speculatively-accessed data* (i.e., read from memory by a speculative instruction) [3], [6], [9], [14], [33], [36]. This is sufficient to block *universal read gadgets* demonstrated in [15], [22], [27], [38], [39] and is of critical importance for the security of general-purpose programs and isolation mechanisms, e.g., kernel code and sandboxes. A recent line of work extends this to protect *non-speculatively-accessed data* [10], [11], [29], [46]. This is primarily useful for hardening cryptographic code, and it incurs more overhead. We consider it out of scope.

B. Speculative Taint Tracking

Speculative Taint Tracking (STT) [45] is a framework designed to prevent the leakage of speculatively-accessed data through microarchitectural covert channels. To achieve this, STT taints speculatively-accessed data, propagates the taint through younger instructions, and delays the execution of those instructions if they are capable of creating a covert channel.

Types of Covert Channels. STT classifies covert channels into two classes: explicit and implicit channels. An *explicit channel* is created by an instruction whose execution time depends on its inputs. Such an instruction is called a *transmitter* (or *transmit instruction*); examples include loads/stores and timing-variable arithmetic. An *implicit channel* is created by an instruction whose inputs influence how or whether subsequent instructions execute (e.g., control-flow instructions). Implicit channels can leak information at *prediction* or at its *resolution* time. For example, a branch instruction can leak information at prediction time (if the predictor contains secrets) or its resolution time (if the branch operand is secret). Finally, STT further distinguishes between *explicit* and *implicit* branches. An explicit branch is a control-flow instruction, while an implicit branch is a conceptual branch that occurs due to hardware mechanisms that change how instructions execute. For example, store-to-load forwarding can be viewed as an implicit branch that checks for an address alias to determine if a load will access the cache.

Taint Propagation and Untainting. STT tracks taint through speculative dataflows using the following taint policy: the input

register of an instruction is tainted if and only if the speculative dataflow into that register is a function of the value returned by an unsafe load. A load is *unsafe* if it returns a secret according to the threat model, and it can dynamically transition to *safe* later. For example, in the *Spectre* model [45], a load is unsafe if and only if there are older unresolved branches. The index in the reorder buffer (ROB) corresponding to the youngest safe load is called the *visibility point* (VP). At a high level, the above is implemented by propagating taint through instructions as they are renamed (i.e., if an instruction's operand is tainted, its output is tainted), and propagating untaint as older loads become safe.

Protection Policy. Given the above, STT blocks *all* covert channels by applying the following protection policy: (1) Explicit Channels are blocked by delaying the execution of transmit instructions until their operands are untainted. (2) Prediction-based Implicit Channels are eliminated by preventing tainted data from affecting the state of any predictor structure. (3) Resolution-based Implicit Channels are eliminated by delaying the effects of branch resolution until the (explicit or implicit) branch's operand becomes untainted.

Youngest Root of Taint. Implementing the untaint operation efficiently is non-trivial as it requires one to re-trace the dataflow from the output of a load through all of its dependents. To mitigate this, STT introduces the *youngest root of taint* (yrot), which tracks the youngest load that a register depends on. The yrot of an instruction is computed as the younger of the yrots of its source operands. The protection policy (above) then treats data as tainted if its yrot is younger than the visibility point.

Putting Everything Together. From the perspective of taint (or yrot) tracking, STT operates as follows:

- **Initial tainting:** The yrot of the destination register of a load is initialized to the ROB index of that load.
- **Taint propagation:** The yrot of an instruction (and its destination register) is set to the younger of its source registers' yrots. The exception is if the instruction is a load, which does not propagate a yrot but instead generates a new one using its own ROB index.
- **Instruction delaying/Untainting:** The execution of a transmit instruction and resolution of a branch is delayed until its yrot becomes older than the VP. Note that branch resolution comes after branch execution. Delaying branch execution is a secure alternative but incurs more overhead.

IV. Motivation: Hardware Complexity of STT

The original STT paper [45] does not analyze the potential impact on logic delay or the area overhead introduced by STT's required hardware changes. This paper revisits the hardware design implied by the original STT proposal—referred to here as OG-STT—and identifies two key limitations that render OG-STT challenging for adoption in high-performance superscalar processors.

Rename Circuit. The rename stage, a critical stage in out-of-order execution, maps architectural registers to physical registers. In superscalar processors, multiple instructions are renamed per cycle—referred to as a *rename group*—with the number of instructions defining the rename width W. The register alias table (RAT) tracks this mapping. Each cycle, free physical registers are allocated to destination operands, while source operands undergo a read-after-write (RAW) dependency check to determine if their physical ID comes from either the same group or the RAT. Additionally, a write-after-write (WAW) dependency check resolves multiple writes to the same architectural register, updating the RAT with the most recent assignment.[1] See Figure 1.

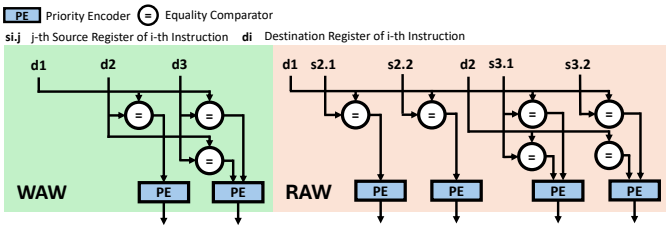


Fig. 1: Read-after-write (RAW) and write-after-write (WAW) dependency checks in rename logic with a rename width $W = 3$. To perform the RAW (or WAW) check, each source (or destination) register is compared against the destination registers of preceding instructions within the same rename group. The comparison results are passed to a priority encoder to identify the most recent match.

Clock Period Increase due to Taint Propagation. OG-STT extends the processor's rename logic to support taint (yrot) propagation. In parallel with the RAT, it introduces a *yrot table* to track the yrots of architectural registers. Each instruction's yrot is computed as the younger yrot of its source operands. If the instruction has a destination register, that register inherits the instruction's yrot—except for the load instruction, whose destination yrot is initialized to the instruction's ROB index.

While this logic works for single-instruction rename per cycle, superscalar processors introduce intra-group dependencies—source yrots may come from the yrot table or earlier instructions in the same rename group. A natural solution is to use a multiplexer to select among all possible source yrots, with selection signals derived from the RAW dependency check. See Figure 2. Since yrots within a rename group are resolved sequentially, the logic delay of yrot propagation grows linearly with the rename width. Consequently, the processor's clock frequency may degrade as the rename width increases, making OG-STT impractical for wide-rename designs.

Area Overhead due to Instruction Delaying. In OG-STT, transmit instructions are delayed in their reservation station (RS), if their yrots are younger than the VP. We call an RS that can store transmitters a *risky RS*. OG-STT augments each risky RS entry with a new field, YRoT, which stores the instruction's yrot. Extra CAM-style comparison logic is introduced to support yrot-aware instruction wake-up. Specifi-

[1] WAW check also identifies the previously assigned physical register, which may be stored in the ROB to support register mapping rollback on squash.

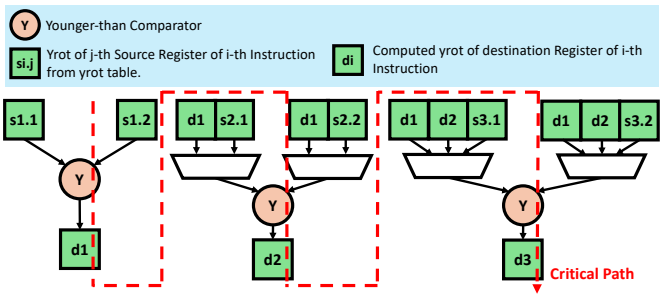


Fig. 2: Yrot propagation circuit of OG-STT with a rename width $W = 3$. The critical path grows linearly with W.

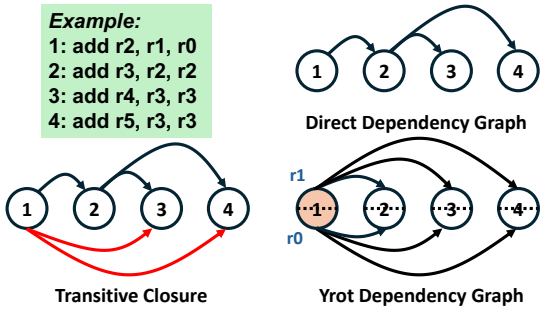


Fig. 3: Dependency graphs between instructions within a rename group.

cally, the VP is broadcast to all risky RS entries and compared against the stored yrots to determine eligibility for wake-up. Consequently, both the number of YRoT fields and the number of younger-than comparators scale linearly with the number of risky RS entries. This not only incurs a substantial area overhead due to the added comparison logic and YRoT fields, but also requires significant wiring complexity for broadcasting VP updates across the entire RS structure.

V. μSTT: Microarchitecture Design for STT

We now present a new microarchitecture design for STT, μSTT, that addresses both limitations discussed in Section IV. This design introduces a shallow taint propagation circuit, the Age Matrix, which uses only a single level of younger-than comparators (Section V-A). As a result, the propagation delay scales more slowly with increasing rename width, making it suitable for wide superscalar designs.

To delay transmit instructions without introducing dedicated VP comparison logic, we introduce a new micro-op, impede, which delays instruction issue via RAW dependencies (Section V-B). Since RAW handling is already supported by existing CAM logic in the reservation station (RS), this mechanism requires no significant modification to the RS.

A. Shallow Taint Propagation Circuit

As discussed in Section IV, the high critical path latency in the taint propagation circuit stems from the fact that the yrot of a later instruction cannot be computed until the yrots of its preceding instructions in the same rename group are resolved. To eliminate this intra-group dependency, we decouple the computation of yrots across instructions renamed in the same cycle. Our approach breaks yrot propagation into two parallel steps that are later merged. First, we build a *yrot dependency graph* to determine which source yrots each instruction depends on—either from the yrot table (for source registers) or from new yrots generated by loads. In parallel, the Age Matrix determines the younger-than relationships among all source yrots in the rename group. Each instruction then selects the youngest of its dependent yrots as its own. For clarity, we first describe this mechanism assuming no load instructions are present, and later extend it to handle loads.

Yrot Dependency Graph. Recall from Section IV that the RAW dependency check in the rename stage identifies which instruction produces a value consumed by a later instruction. As illustrated in Figure 3, instruction 2 produces r3, which is subsequently used by instructions 3 and 4. These producer-consumer relationships form a direct dependency graph, with edges from instruction 2 to 3 and 4. Unlike the rename logic, yrot tracking must trace back to the original source register that has no dependency within the same rename group. We refer to such registers as *free registers*.

To identify the yrot sources for each instruction, we compute the transitive closure of the direct dependency graph. This reveals all instructions that an instruction's yrot may depend on. Connectivity is checked recursively, as shown in Algorithm 1. Two instructions are considered connected if they are either directly linked or transitively connected through intermediate instructions. Note that every instruction connects to itself, which is omitted in Algorithm 1 and Figure 3. The recursion terminates when two instructions are adjacent (no intermediate instructions) and returns their connectivity according to the direct dependency graph. The hardware depth of this transitive closure computation is logarithmic in the rename width, because at each layer, the maximum distance covered doubles (e.g., 1-hop, 2-hop, 4-hop, etc.).

We then build the yrot dependency graph by extending transitive closure edges to source registers and pruning edges from non-free registers, whose yrots are fully derived from others. As shown in Figure 3, the yrots of all four instructions depend on two free registers: r1 and r0.

Age Matrix. The Age Matrix takes a vector of $2W$ source yrots (two source registers per instruction in a rename group of width W) and computes pairwise younger-than relationships. We denote two source yrots of the i-th instruction as $yrot_{2i-1}$ and $yrot_{2i}$ ($i \in [1, W]$). Figure 4a illustrates the structure of the Age Matrix, which has a shape of $2W \times 2W$. Each row and column index corresponds to a source yrot, and each element in the matrix encodes the younger-than relationship between the yrots at those indices. For example, the orange cell $AgeMatrix[5, 1]$ in Figure 4a is set to 1 if the first source operand of the third instruction ($yrot_5$) is younger than the first source operand of the first instruction ($yrot_1$). Since the matrix is antisymmetric—the lower triangular part is the logical inverse (flip) of the upper triangular—only the upper triangular portion requires actual younger-than comparators,

Algorithm 1: Connectivity between two instructions.

Input : i, j: Instruction indices
$dgraph[W][W]$: Direct dependency graph
Output: 1 (connected) / 0 (disconnected)

Function `CheckCon` $(i, j, dgraph)$**:**
  $dis \leftarrow j - i$;
  if $dis == 1$ **then**
    return $dgraph[j][i]$;
  else
    for $k \leftarrow 1$ **to** $dis - 1$ **do**
      $bridge1 \leftarrow$ `CheckCon`$(i, i + k, dgraph)$;
      $bridge2 \leftarrow$ `CheckCon`$(i + k, j, dgraph)$;
      $inter\_con[k - 1] \leftarrow bridge1 \wedge bridge2$;
    return $dgraph[j][i] \vee (\bigvee_k inter\_con[k])$;

which amounts to $W(2W - 1)$ comparator instances.

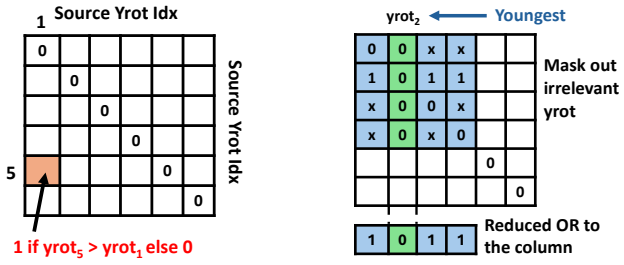
Yrot Selection Logic. The Age Matrix can be further processed to identify the youngest yrot for each instruction. Take instruction 2 as an example, illustrated in Figure 4b. This instruction may depend on the yrot sources of the first two instructions—namely $yrot_1$ through $yrot_4$—forming the blue square region in the matrix. Suppose $yrot_2$ is the youngest among these. The column corresponding to $yrot_2$ (column index 2) will be filled with all 0s, since no other yrots are younger than it. At the same time, the row corresponding to $yrot_2$ will have all 1s, except at the diagonal entry (which compares it with itself). In contrast, the columns associated with other yrots will have at least one bit-1, indicating the presence of a younger yrot. By applying an `OR`-reduction to each column, only the column corresponding to the youngest yrot will reduce to 0, which can then be used to select the appropriate yrot from the source yrots vector.

Not all source yrots are relevant to a given instruction. For example, if $yrot_2$ is not a dependency of instruction 2, it should be excluded from the selection process. To mask out irrelevant yrots, we apply two masks for each element at row i and column j. This is summarized in Equation (1).

$$MaskOut[i][j] = \neg DepGraph[2][j] \vee \big(AgeMatrix[i][j] \wedge DepGraph[2][i]\big) \quad (1)$$

First, $\sim DepGraph[2][j]$ forces the columns of irrelevant yrots to all 1s, ensuring they are not selected (i.e., their `OR`-reduction will be 1). Second, $DepGraph[2][i]$ zeros out the rows of irrelevant yrots, ensuring other candidates are not mistakenly blocked.

Adding Loads. The previous procedure assumes no load instructions, but loads change the yrot propagation behavior because they generate new yrots and terminate the propagation path—the destination yrot is independent of source operands. To prevent yrot propagation through loads, we adjust the direct dependency graph by removing all edges originating from load instructions. As shown in Figure 5, when instruction 2 is a load, its outgoing edges are removed, making instructions 3 and 4 have free registers. However, their source yrots should now come from instruction 2's ROB index, not the yrot table. To correctly retrieve source yrots, instruction i in the decode group requires two $i : 1$ multiplexers, to select between the yrot table entry or the ROB index of the corresponding load instruction in the rename group.



(a) (b)

Fig. 4: (a) Age matrix structure and semantics. We denote two source yrots of the i-th instruction in the rename group as $yrot_{2i-1}$ and $yrot_{2i}$ ($i \in [1, W]$). (b) Yrot selection example for instruction 2, combining the Age Matrix with the yrot dependency graph ($DepGraph$).

Fig. 5: Adjust the dependency graph when handling load instructions in the rename stage.

Younger-than Comparator. The original STT paper uses a simple `max` function to select the younger yrot between two source yrots. However, it does not explain how the `max` is implemented in the context of the circular ROB, where indices wrap around. As shown in Figure 6 (left), if two yrots are on the same side of the ROB head, the larger index is younger. In contrast, if two yrots lie on opposite sides of the head—i.e., across the wrap-around—the smaller index is younger. Prior work [31] addressed this ambiguity with three greater-than operations: two to determine relative positions to the head and one for index comparison.

Since the number of younger-than comparators in the Age Matrix grows quadratically with rename width and lies on the critical path, reducing comparator complexity is the key to minimize delay and area overhead. As shown in Figure 6 (right), we improve on this by unrolling the ROB once into two virtual regions. We append one extra bit, called the `wrap_bit`, to the top of ROB tail. This bit toggles each time the tail crosses a virtual region boundary. Yrots generated by load instructions inherit this `wrap_bit` from the ROB tail when assigned. Because valid yrots always lie between the ROB head and tail, they span at most two virtual regions. Thus, if two yrots are in the same region, they are also on the same side of the ROB head, and the one with the larger index

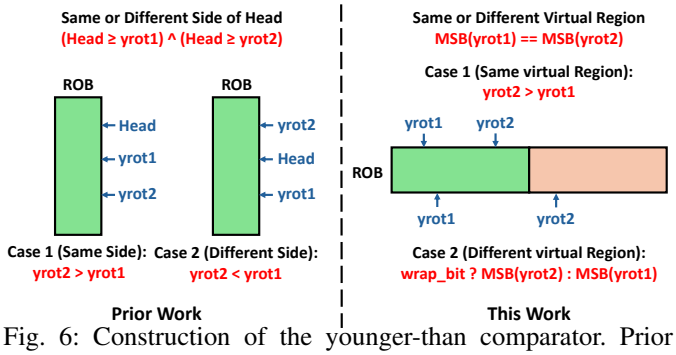


Fig. 6: Construction of the younger-than comparator. Prior works require three greater-than operations, whereas our design requires only one.

is younger. If they fall in different regions, the yrot in the wrap-around region—i.e., the one sharing the `wrap_bit` of the ROB tail—is considered younger. With this encoding, the younger-than comparator requires only a single greater-than operation, reducing both logic complexity and delay.

B. Instruction Delaying with Impede Micro-ops

As discussed in Section IV, extending the reservation station (RS) to support VP-yrot comparisons incurs substantial area overhead. To avoid this, we introduce a new micro-op called *impede*. The impede micro-op delays the issue of a transmitter by acting as a producer of the transmitter's input operand. It also takes the transmitter's yrot as an input operand and stalls in the RS until this yrot crosses the VP and is broadcast via the common data bus (CDB).

Speculation Epoch. In OG-STT, each load assigns its destination yrot as its ROB index. This design results in many distinct inflight yrots. In the impede micro-ops design, each distinct yrot must eventually be broadcast over the common data bus (CDB), which can lead to significant communication overhead. To mitigate this, we observe that loads between two speculation initiators (e.g., branches) become non-speculative together. We propose grouping these loads together and assigning them a common yrot. We refer to this shared yrot as a speculation epoch, and it is represented by the ROB index of the corresponding speculation initiator.

Impede Micro-op. Transmit instructions, denoted as $\mathsf{ry} \leftarrow \mathsf{transmitter}(\mathsf{rx})$, are decomposed into two micro-ops. The first is the impede micro-op, which takes two input operands, as shown in Equation (2). One operand is the original transmitter input, rx, and the other is the yrot of the transmitter, $\mathsf{yrot}(\mathsf{transmitter})$. Importantly, yrots share the same value space as physical register IDs and are stored in the same field within the RS entry. To distinguish between a yrot and a physical register ID, the field must be extended by at least one bit (or more, if yrots require greater bit width). Functionally, the impede micro-op simply forwards its input rx to a new destination rx'.

$$\mathsf{rx}' \leftarrow \mathsf{impede}(\mathsf{yrot}(\mathsf{transmitter}), \mathsf{rx}) \tag{2}$$

$$\mathsf{ry} \leftarrow \mathsf{transmitter}(\mathsf{rx}') \tag{3}$$

The second micro-op replicates the behavior of the original transmit instruction and uses rx' (produced by the impede micro-op) as its input.

While the two micro-ops per transmitter are installed in the RS, a *VP pump* broadcasts speculation epochs, as they cross the VP, on the CDB. When this broadcasted epoch matches the yrot operand of an impede micro-op, the micro-op becomes ready to issue (assuming its other operand is also ready), and subsequently wakes up the corresponding transmitter micro-op (after being issued). In this design, the yrot is treated as a source operand of the impede micro-op. As a result, standard wake-up and writeback logic—already used for handling RAW dependencies—can be leveraged to delay the impede micro-op, and consequently the transmitter, until the yrot crosses the VP and is broadcast on the CDB.

Implicit Channel Treatment. A straightforward way to apply impede micro-ops to protect branches is to treat branches as transmit instructions and to decompose each into two micro-ops, following the strategy used for other transmitters. However, this approach introduces two drawbacks: it unnecessarily delays branch execution, increases the number of impedes inserted into the instruction stream, and adds an extra cycle of issue delay (assuming the impede micro-op takes one cycle to execute and writeback).

We propose delaying branch resolution until it becomes the oldest unresolved speculation initiator. At that point, it is safe to resolve the branch because no older speculative loads exist, ensuring that the input is untainted. This is similar to, but more conservative than, the original STT design, which resolves branches as soon as their inputs are untainted.

This approach offers two key benefits. First, only mispredicted branches incur a delay. We note that most branches are predicted correctly due to modern high-accuracy branch predictors. Second, delaying speculative branch resolution ensures that no speculative instructions remain in flight after a squash. As a result, the yrot table can be safely reset without requiring checkpoints for rollback, reducing area overhead in the rename stage. While we explained this idea in the context of explicit branches (control-flow instructions), it can also be applied to other implicit channels.

Deadlock Treatment. The above design suffers from a potential deadlock scenario, where dispatched transmitters miss their corresponding yrot broadcast on the CDB and are permanently stalled. For example, a transmit instruction may be assigned a yrot that has already been broadcast before the instruction is dispatched. Although this yrot is no longer speculative, the broadcast is not replayed, and missing it can leave the associated impede micro-op permanently stalled—potentially leading to deadlock. To prevent this, yrots must be explicitly invalidated once they are broadcast. When a yrot is sent on the CDB, any matching entries in the yrot table are marked as untainted (i.e., invalid). In addition, dispatched impede micro-ops must compare their yrot operand against the current VP. If the yrot has already crossed the VP, the yrot operand is marked as ready, allowing the impede micro-op to proceed

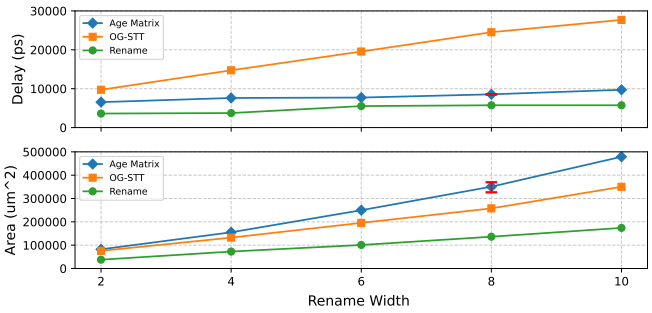


Fig. 7: Critical path delay and area consumption of three designs (RENAME, OG-STT, and AGEMATRIX) as the rename width varies from 2 to 10. The red error bar at rename width 8 illustrates the impact of varying yrot width from 9 to 11 bits.

without waiting for a broadcast.

VI. EVALUATION

We evaluate our Age Matrix-based taint propagation circuit and the impede micro-op instruction delaying scheme. Our evaluation uses parameters in Table I, which are derived from prior reverse engineering efforts targeting Apple's Firestorm microarchitecture [18], [30], a representative large-window, high-performance design that poses significant challenges for supporting STT.

A. Taint Propagation Delay and Area

Experimental Setup. We implement the standalone rename-stage taint propagation circuit in Verilog and synthesize it using Cadence VLSI tools with the open-source Skywater 130nm technology process [1]. We compare three designs: the baseline rename circuit (RENAME), the OG-STT design (OG-STT), and our Age Matrix-based design (AGEMATRIX). Each taint propagation circuit includes both taint-specific logic and baseline rename logic.

Results. We start from the default parameter settings and vary the rename width from 2 to 10. Figure 7 shows the logic delay and area consumption for the three designs. As the results show, AGEMATRIX scales significantly better than OG-STT in terms of delay. As the rename width increases, the delay gap between OG-STT and both AGEMATRIX and RENAME widens considerably. On average across all tested rename widths, the latency of the AGEMATRIX is 1.69× that of the RENAME, while OG-STT incurs a 3.86× latency overhead. In terms of area consumption, the AGEMATRIX incurs 25% more area than OG-STT and 142% more than the RENAME. At the default rename width of 8, AGEMATRIX removes 85% latency overhead of OG-STT with a 36% area increase.

We further investigate the impact of yrot bit width on both delay and area. Specifically, we vary the yrot width from 9 to 11 bits, corresponding to ROB sizes of 256 to 1024, with rename width fixed at 8. The results are shown as red error bars in Figure 7. As the results show, yrot width has a negligible impact on critical path delay and only a modest effect on area.

B. Performance Overhead of Instruction Delaying

Simulation Setup. We use the Gem5 simulator [8] to evaluate the performance impact of introducing impede micro-ops on an AArch64 architecture. We run SPEC CPU2017 benchmarks [2] with the reference input set and apply SimPoint analysis [34]. The program execution is divided into chunks of 100 M instructions, and up to 30 representative SimPoints per application are selected using a clustering algorithm. Each SimPoint is preceded by a 10 M instructions warm-up phase. Final performance results for each application are computed as the weighted sum across all selected SimPoints.

TABLE I: Simulated Architecture Parameters

HW Components	Parameters
Architecture	1 core at 3.2GHz, AArch64 ISA
Core	8 fetch/decode/rename/dispatch/issue/writeback/commit, 130/60 LQ/SQ entries, 330 ROB entries, 380 integer physical registers, 128 flags physical registers, 120 instruction queue entries, TAGE branch predictor
L1 I-Cache	192KB, 6-way, 2-cycle latency
L1 D-Cache	128KB, 8-way, 2-cycle latency
L2 Cache	12MB, 12-way, 20-cycle latency
DRAM	20ns latency after L2

Configurations. In our Gem5 implementation, we adopt the Spectre model [45], where a load instruction is considered unsafe until there are no older unresolved branches. For leakage channels, we assume that load and store instructions can create explicit channels.[2] Meanwhile, branch instructions are assumed to create implicit channels. We evaluate four different designs: (1) DELAYACC [32] delays the execution of unsafe loads. (2) DELAYEXEC [31] (which models a naïve taint-based scheme) delays the execution of tainted loads/stores and the resolution of tainted branches until there is no unresolved branch older than them. (3) OG-STT [45] delays the execution of loads/stores and the resolution of branches until their yrots cross the VP. (4) Our proposal IMPEDEMEM-DBR applies impede micro-ops to loads/stores. A branch is protected by delaying its resolution until it is the oldest unresolved branch. Finally, UNSAFE is the insecure baseline.

Main Performance Results. Figure 8 compares the execution time of four configurations: OG-STT, IMPEDEMEM-DBR (our proposal), DELAYEXEC, and DELAYACC. Their performance overheads relative to the insecure baseline (UNSAFE) on average are 16%, 21%, 32%, and 60%, respectively. These results highlight the importance of adopting STT-style taint propagation and untainting mechanisms. OG-STT outperforms the naïve tainting in DELAYEXEC by 16 percentage points, and the non-taint-based defense DELAYACC by 44 percentage points. Our proposal, IMPEDEMEM-DBR, achieves similar benefits to OG-STT with only 5 additional percentage points overhead—while requiring significantly fewer hardware modifications.

Optimizations over the Impede Scheme. We evaluate the performance impact of two key optimizations added to the

[2]Although a store does not commit its data to memory until retirement, it initiates address translation during execution, thereby transmitting its address operand through the page walk side channel [38].

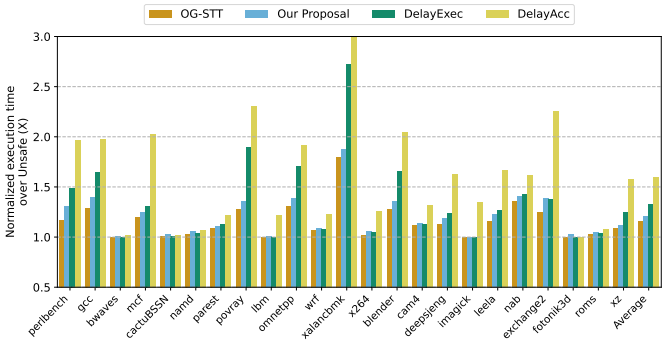


Fig. 8: Execution time (normalized to UNSAFE) of SPEC CPU2017 benchmarks with the configurations OG-STT, DELAYEXEC, DELAYACC and our proposal IMPEDEMEM-DBR.

impede micro-op scheme: the use of speculation epochs and delayed branch resolution. Without speculation epochs, the performance overhead increases significantly—from 21% to 38%—due to a larger number of yrots consuming CDB bandwidth, as described in Section V-B. The delayed branch resolution optimization is also notably more effective than applying impede micro-ops directly to branches. Specifically, it reduces performance overhead from 32% to 21%.

Sensitivity Study. Finally, we perform a sensitivity study over three architecture parameters that may contribute to resource contention introduced by impede micro-ops: writeback width, integer functional unit count, and instruction queue size.[3] We evaluate the impact of these parameters on performance for three configurations: UNSAFE, OG-STT, and our proposal. Table II reports the results. Our proposal is most sensitive to the writeback width. Reducing the writeback width from 8 to 4 leads to a 16% increase in overhead for our design, compared to only 3% for UNSAFE and 5% for OG-STT. We remark that our proposal is primarily applicable to large-window designs, where the writeback width is typically larger than 4.

TABLE II: Sensitivity Study.

	Writeback Width			Functional Unit Count			Instruction Queue Size		
	4	6	8	2	4	6	40	80	120
Unsafe	3%	0%	0%	3%	0%	0%	6%	0%	0%
OG-STT	21%	17%	16%	19%	16%	16%	20%	16%	16%
Our proposal	37%	24%	21%	33%	21%	21%	27%	22%	21%

VII. RELATED WORK

This paper presents the first in-depth implementation study of speculative taint tracking (STT). The most relevant prior work, Secure-by-Construction [17], mitigates the logic delay of taint propagation by offloading it from the rename stage to the execution stage. A dedicated functional unit is instantiated to compute the yrot of a destination register when its associated instruction reaches the execution stage. While this solves logic delay in the rename stage, it does not address the area overhead of instruction delaying. The reservation station still requires per-instruction YRoT fields and younger-than comparators for wake-up logic. Assuming a ROB size of 512 entries, there can be up to 512 in-flight instructions. Each instruction requires three yrot copies (two for source registers, one for the destination), leading to a total of 512×3 9-bit yrot registers. We estimate the area overhead for just yrot state bits to be $4.3 \times 10^5 \mu\text{m}^2$ (not including comparator logic or extended CDB), which exceeds the area of the entire 8-wide rename-stage Age Matrix circuit (whose area is $3.5 \times 10^5 \mu\text{m}^2$).

[3]Impede micro-ops are executed on integer functional units. Other parameters (e.g. physical register file size and ROB size) have similar effect. We do not put them here to save space.

VIII. CONCLUSION

This paper proposes a novel microarchitectural design for Speculative Taint Tracking (STT), μSTT, that addresses two key limitations of the original design: (1) non-scalable logic delay of the taint propagation circuit, and (2) area overhead associated with the instruction delaying mechanism. Specifically, the proposed Age Matrix enables scalable taint propagation, while the impede micro-ops provide an instruction delaying mechanism that requires minimal hardware modification. Together, these innovations offer a practical path toward realizing STT in modern superscalar processors.

REFERENCES

[1] "SkyWater SKY130 PDK," https://skywater-pdk.readthedocs.io/en/main/.
[2] "SPEC CPU2017," https://www.spec.org/cpu2017.
[3] P. Aimoniotis, A. B. Kvalsvik, X. Chen, M. Själander, and S. Kaxiras, "Recon: Efficient detection, management, and use of non-speculative information leakage," in *MICRO'23*.
[4] S. Ainsworth, "Ghostminion: A strictness-ordered cache system for spectre mitigation," in *MICRO'21*.
[5] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *IACR'18*.
[6] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels," in *PACT'19*.
[7] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting Speculative Execution through Port Contention," in *CCS'19*.
[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *ACM SIGARCH Comput. Archit. News*, 2011.
[9] C. Carruth, "Cryptographic Software in a Post-Spectre World," https://www.youtube.com/watch?v=uiN-Wmtk1aU&t=4350s, 2020.
[10] R. Choudhary, A. Wang, Z. N. Zhao, A. Morrison, and C. W. Fletcher, "Declassiflow: A static analysis for modeling non-speculative knowledge to relax speculative execution security measures," in *CCS'23*.
[11] R. Choudhary, J. Yu, C. Fletcher, and A. Morrison, "Speculative privacy tracking (spt): Leaking information from speculative execution without compromising privacy," in *MICRO'21*.
[12] M. Dai, R. Paccagnella, M. Gomez-Garcia, J. McCalpin, and M. Yan, "Don't mesh around: Side-channel attacks and mitigations on mesh interconnects," in *USENIX Security'22*.
[13] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *USENIX Security'18*.
[14] C. Green, C. Nelson, M. Thottethodi, and T. Vijaykumar, "Safebet: Secure, simple, and fast speculative execution," *arXiv'23*.
[15] M. Hertogh, S. Wiebing, and C. Giuffrida, "Leaky address masking: Exploiting unmasked spectre gadgets with noncanonical address translation," in *IEEE S&P'24*.
[16] J. Horn, "Speculative execution, variant 4: speculative store bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[17] T. Jauch, A. Wezel, M. R. Fadiheh, P. Schmitz, S. Ray, J. M. Fung, C. W. Fletcher, D. Stoffel, and W. Kunz, "Secure-by-construction design methodology for cpus: Implementing secure speculation on the rtl," in *ICCAD'23*.

[18] D. Johnson, "Apple microarchitecture research," https://dougallj.github.io/applecpu/firestorm.html, 2021.

[19] J. Kim, J. Chuang, D. Genkin, and Y. Yarom, "Flop: Breaking the apple m3 cpu via false load output predictions," in *USENIX Security'25*.

[20] J. Kim, D. Genkin, and Y. Yarom, "Slap: Data speculation attacks via load address prediction on apple silicon," in *IEEE S&P'25*.

[21] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv'18*.

[22] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE S&P'19*.

[23] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *WOOT'18*.

[24] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *USENIX Security'17*.

[25] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," in *ASPLOS'96*.

[26] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE S&P'15*.

[27] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv'19*.

[28] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring(s): Side channel attacks on the cpu on-chip ring interconnect are practical," in *USENIX Security'21*.

[29] M. Patrignani and M. Guarnieri, "Exorcising spectres with secure compilers," in *CCS'21*.

[30] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "Pacman: attacking arm pointer authentication with speculative execution," in *ISCA'22*.

[31] M. Sabbagh and Y. Fei, "Secure speculative execution via risc-v open hardware design," in *CARRV'21*.

[32] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction," in *ISCA'19*.

[33] P. Schmitz, T. Jauch, A. Wezel, M. R. Fadiheh, T. Tiemann, J. Heller, T. Eisenbarth, D. Stoffel, and W. Kunz, "Okapi: Efficiently safeguarding speculative data accesses in sandboxed environments," *arXiv'23*.

[34] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS'02*.

[35] A. Tatar, D. Trujillo, C. Giuffrida, and H. Bos, "TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering," in *USENIX Security'22*.

[36] M. Vassena, C. Disselkoen, K. v. Gleissenthall, S. Cauligi, R. G. Kıcı, R. Jhala, D. Tullsen, and D. Stefan, "Automatically eliminating speculative leaks from cryptographic code with blade," in *POPL'21*.

[37] J. Wan, Y. Bi, Z. Zhou, and Z. Li, "MeshUp: Stateless cache side-channel attack on cpu mesh," in *IEEE S&P'22*.

[38] A. Wang, B. Chen, Y. Wang, C. Fletcher, D. Genkin, D. Kohlbrenner, and R. Paccagnella, " Peek-a-Walk: Leaking Secrets via Page Walk Side Channels ," in *IEEE S&P'25*.

[39] S. Wiebing, A. de Faveri Tron, H. Bos, and C. Giuffrida, "Inspectre gadget: Inspecting the residual attack surface of cross-privilege spectre v2," in *USENIX Security'24*.

[40] Y. Yarom and K. Falkner, "Flush+reload: a high resolution, low noise, L3 cache side-channel attack," in *USENIX Security'14*.

[41] T.-Y. Yeh and Y. Patt, "Alternative implementations of two-level adaptive branch prediction," in *ISCA'92*.

[42] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data oblivious ISA extensions for side channel-resistant and high performance computing," in *NDSS'19*.

[43] J. Yu, T. Jaeger, and C. W. Fletcher, "All your PC are belong to us: Exploiting non-control-transfer instruction BTB updates for dynamic PC extraction," in *ISCA'23*.

[44] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution," in *ISCA'20*.

[45] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. Fletcher, "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data," in *MICRO'19*.

[46] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, "Ultimate SLH: Taking speculative load hardening to the next level," in *USENIX Security'23*.

[47] Z. Zhang, M. Tao, S. O'Connell, C. Chuengsatiansup, D. Genkin, and Y. Yarom, "Bunnyhop: Exploiting the instruction prefetcher," in *USENIX Security'23*.