ReCon: Efficient Detection, Management, and Use of Non-Speculative Information Leakage

Pavlos Aimoniotis
pavlos.aimoniotis@it.uu.se
Uppsala University
Sweden

Amund Bergland Kvalsvik
amund.kvalsvik@ntnu.no
Norwegian University of
Science and Technology
Norway

Xiaoyue Chen
xiaoyue.chen@it.uu.se
Uppsala University
Sweden

Magnus Själander
magnus.sjalander@ntnu.no
Norwegian University of
Science and Technology
Norway

Stefanos Kaxiras
stefanos.kaxiras@it.uu.se
Uppsala University
Sweden

ABSTRACT

In a speculative side-channel attack, a secret is improperly accessed and then leaked by passing it to a transmitter instruction. Several proposed defenses effectively close this security hole by either delaying the secret from being loaded or propagated, or by delaying dependent transmitters (e.g., loads) from executing when fed with *tainted* input derived from an earlier speculative load. This results in a loss of memory-level parallelism and performance.

A security definition proposed recently, in which data already leaked in non-speculative execution need not be considered secret during speculative execution, can provide a solution to the loss of performance. However, detecting and tracking non-speculative leakage carries its own cost, increasing complexity. The key insight of our work that enables us to exploit non-speculative leakage as an optimization to other secure speculation schemes is that the majority of non-speculative leakage is simply due to pointer dereferencing (or base-address indexing) — essentially what many secure speculation schemes prevent from taking place speculatively.

We present ReCon that: i) efficiently detects non-speculative leakage by limiting detection to pairs of directly-dependent loads that dereference pointers (or index a base-address); and ii) piggybacks non-speculative leakage information on the coherence protocol. In ReCon, the coherence protocol remembers and propagates the knowledge of what has leaked and therefore what is safe to dereference under speculation. To demonstrate the effectiveness of ReCon, we show how two state-of-the-art secure speculation schemes, Non-speculative Data Access (NDA) and speculative Taint Tracking (STT), leverage this information to enable more memory-level parallelism both in a single core scenario and in a multicore scenario: NDA with ReCon reduces the performance loss by 28.7% for SPEC2017, 31.5% for SPEC2006, and 46.7% for PARSEC; STT with ReCon reduces the loss by 45.1%, 39%, and 78.6%, respectively.

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3623770

CCS CONCEPTS

• **Computer systems organization** → **Superscalar architectures**; • **Security and privacy** → **Hardware-based security protocols**.

KEYWORDS

Speculation, side-channels, load pair, non-speculative leakage

ACM Reference Format:
Pavlos Aimoniotis, Amund Bergland Kvalsvik, Xiaoyue Chen, Magnus Själander, and Stefanos Kaxiras. 2023. ReCon: Efficient Detection, Management, and Use of Non-Speculative Information Leakage. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3613424.3623770

1 INTRODUCTION

Since the discovery of speculative side-channel attacks [24, 29], a wide variety of transient execution attacks have been found [10, 22, 23, 25, 30, 32, 41]. These attacks vary in attack method, being able to leak information through port contention [10], micro-op caches [35], and reorder buffer contention [3] and even to break the ARM Pointer Authentication [32]. To mitigate against speculative side-channel attacks, a slew of speculative execution defenses have been proposed [4, 5, 14, 28, 38, 39, 47, 52, 53, 56]. These defenses differ in the threat model they operate under, their performance overhead compared to an unsafe baseline, and the amount of modifications they introduce to the system.

Early works, such as InvisiSpec[53] and Ghost Loads[38], as well as later proposals such as Muontrap [5], provide solutions to block speculative side-channel attacks in the cache hierarchy, but do not eliminate many other side-channels [9].

Later works, such as Non-speculative Data Access (NDA) [52], and Speculative Taint Tracking (STT) [56] improve on previous work by securing more speculative side-channels. Their common characteristic is that they focus on delaying potentially dangerous, speculative *transmitter* instructions. STT outperforms NDA through a method called taint tracking, in which potential secrets are tracked, and only possible transmitters are delayed. Still, STT is unable to issue load-dependent instructions, since the second dependent load is considered a transmitter. This results in a loss of instruction- and memory-level parallelism and a corresponding loss in performance.

To recover some of the performance overhead that these defense proposals introduce, several optimizations have been proposed. They either rely on compiler support [48, 58], or modifications of the core microarchitecture and of the memory hierarchy [4, 55]. We take a different approach to the optimization of secure speculation schemes by proposing to efficiently detect non-speculative leakage and use this information to lift security mechanisms for safe speculative loads.

Our approach is based on a new security definition concerning the exposure of secrets, proposed by Choudhary et al. in Speculative Privacy Tracking (SPT) [14]. Under this definition, "*any data that can leak through the program's non-speculative execution should not be treated as secret during the program's speculative execution*" [14].

SPT leverages this security definition to provide comprehensive mitigation for speculative side-channels. More specifically, SPT proposes continuous taint tracking that spans all execution (both non-speculative and speculative) to detect and track non-speculative leakage. Assisted by a sophisticated forward and backward untaint mechanism, SPT maximizes the amount of non-speculative leakage detected. However, this requires changes of significant complexity in the core. Moreover, SPT protects secrets loaded in registers prior to speculative execution, albeit at a relatively steep performance cost. Register protection, however, is not a requirement to eliminate *universal read gadgets* that can leak all memory [14], which is the main focus of our work. Besides, non-speculatively accessed secrets can be protected by other means [54], which renders such high-cost register protection mechanism less appealing for our purposes.

Building on the SPT security definition, we make the following observation: A large part of the information leakage prevented by secure speculation schemes, such as STT and NDA, simply comes from *load pairs that dereference pointers or index base-addresses.* In the interest of conciseness, for the remainder of this paper, we will liberally use the term **pointer** to refer to either an (8-byte) address loaded from memory or an (8-byte) integer index that is loaded from memory and added to a (constant) base address. These two cases are indistinguishable in our approach. Similarly, we use the gerund **dereferencing** to encompass both pointer dereferencing and base-address indexing.

Moreover, trying to prevent exactly this type of leakage is what causes a significant performance loss in secure schemes. While program execution may leak (non-speculatively) in many different ways and through many different side-channels, we focus exclusively on exploiting the non-speculative leakage due to load pairs.

The focus on load pairs enables us to distill the detection and management of non-speculative leakage (that is highly complex in the general case) into two simple functions: i) observe a load pair dereferencing pointers, and ii) mark this particular pointer's *value* as safe to dereference under speculation. If two loads execute non-speculatively, where the second load dereferences the value of the first load, then our approach, called ReCon, **reveals** the value of the first load as *public*. If a value has not been revealed or is changed, it must be **concealed** as secret. In ReCon, we mark the *address* that contains a revealed or concealed value, correspondingly, as such.

Consider the following example:

```
// Non-speculative execution
// Address 0x13 may contain a secret
PC1: load  r1, [0x13]
PC2: load  r2, [r1]
// but the secret at 0x13 is now revealed

// Speculative execution
if(cond){
    PC3: load  r3, [0x13]
    // It is safe to pass the revealed
    // value at 0x13 to a transmitter
    PC4: load  r4, [r3]
}
```

Here, PC2 dereferences the value of the PC1 load (i.e., uses the value stored at address 0x13 as an address), revealing this information to the memory hierarchy. Committing this non-speculative access (PC2), as part of the intended program execution, means that the value stored at 0x13 cannot be considered a secret any longer. ReCon marks the memory location 0x13 as *revealed.* This holds true until the value at 0x13 is updated by a store instruction, in which case, the memory location 0x13 would be marked as *concealed.* Speculatively reading from the *revealed* memory location 0x13 (e.g., by the PC3 load), means that there is no need to apply speculative defenses and the dependent PC4 load can dereference the value and execute without leaking anything that is not already public.

A further insight that drives our work concerns the preservation and propagation of the conceal/reveal information throughout the cache hierarchy: it is possible to propagate the information efficiently via the coherence protocol. We tag cache lines with reveal/conceal information, and we piggyback this information on the coherence transactions of a standard directory MESI protocol (or similar). Coherence handles the updates to the value of the pointer, at the same time resetting the leakage information as needed. We explain this in Section 5.3 where we show that without modifications to the base coherence protocol (only adding conceal/reveal information on top of it), we can effectively manage leakage information throughout a multicore cache hierarchy for both single- and multi-threaded workloads.

The overall approach, called ReCon (*Reveal/Conceal*), is an efficient, low-complexity technique for detecting *revealed* addresses from non-speculative pointer dereferencing, for tracking revealed addresses throughout a multicore cache hierarchy, and for optimizing secure speculation schemes (Section 4 and Section 5).

In this work, we apply ReCon on STT and NDA and evaluate the resulting schemes with benchmarks from the SPEC2017 and SPEC2006 benchmark suites (Section 6). Our results show that ReCon reduces the overhead over the unsafe baseline from 13.2% to 9.4% and from 8.9% to 4.9% on SPEC2017, and from 10.4% to 7.2% and from 8.1% to 5% on SPEC2006, when applied on-top of NDA and STT, respectively. We also evaluate parallel benchmarks from the PARSEC benchmark suite, and we observe a 46.7% and 78.6% reduction in the overhead incurred in total execution time, for NDA and STT, respectively.

2 BACKGROUND

In this section, we describe the previous work that ReCon leverages to enable safe execution of loads that access revealed addresses.

2.1 Non-speculative Data Access (NDA)

Non-speculative Data Access (NDA) [52] defends by blocking the propagation of secrets at the earliest possible stage. While NDA proposes several variations of its central defense mechanism, we focus on the strategy labeled *permissive propagation*.

In permissive propagation, potential secrets can be acquired by speculative load instructions, but the secrets are not allowed to propagate to dependent instructions, i.e., broadcast is delayed until the originating instruction is non-speculative. Hence, there is no possible way to expose potential secrets. NDA does not require any extra modification to handle either explicit or implicit channels, as the potential secrets are never released into the rest of the processor core in any way. This is in contrast to STT [56] and SPT [14], which have to use taint tracking to keep track of the propagation of potential secrets to dependent instructions, and need to handle both explicit and implicit channels when inputs are tainted.

While NDA limits instruction level parallelism by not propagating the results of load instructions, it achieves the same amount of memory level parallelism as STT [56] (dependent loads are blocked in both) while proposing a far simpler scheme. However, blocking all non-transmitting instructions that depend on speculative load values incurs larger performance penalties.

2.2 Speculative Taint Tracking (STT)

Speculative Taint Tracking (STT) [56] is a state-of-the-art safe speculative execution scheme due to its performance and versatility. STT relies on two fundamental principles: Firstly, all instructions that do not have a dependency originating from a load instruction are allowed to execute as normal, including loads. Second, it delays speculative transmitting operands whose execution depends on a speculatively loaded value until the value is confirmed to be non-speculative and therefore not a secret.

STT uses a taint tracking mechanism, similar to dynamic information flow tracking (DIFT) [45], to prevent the execution of transmitting instructions that depend on speculatively loaded values. STT taints the output register of a speculative load instruction, or any instruction dependent on tainted data. STT automatically untaints the destination register and all tainted registers that originate from this destination register, as soon as the corresponding load instruction becomes safe, i.e., when it becomes non-speculative.

STT also provides an extensive analysis of *explicit* and *implicit* channels, which respectively, directly and indirectly, can be used to leak secrets, such as in some Spectre variants [24] (explicitly) and SmotherSpectre [10] (implicitly). To prevent the use of explicit channels, STT delays the execution of any transmit instruction whose operands are tainted. This means that any instruction with tainted input will not be executed until its inputs are no longer tainted. To prevent the use of implicit channels, STT ensures that the program control flow is not influenced by tainted data. This means that branch predictions can still occur as normal, but the resolution of branch predictions, whether correctly or wrongly predicted, is delayed until the branch inputs are untainted.

2.3 Speculative Privacy Tracking (SPT)

Speculative Privacy Tracking (SPT) [14] is a defense mechanism that offers even more comprehensive protection against speculative side-channels, compared to NDA and STT. SPT shares similarities with STT, as they are both schemes based on DIFT [45], however, SPT protects all leakage under speculation, including register values that were set pre-speculation and have not leaked their values (non-speculative secrets). SPT refers to the taint tracking mechanism of STT as *s-taint*, to differentiate it from their proposed taint tracking mechanism. While s-taint focuses on speculative tracking through registers, SPT proposes a global, continues taint tracking mechanism that also propagates taint/untaint information through the memory hierarchy. It also introduces novel ideas to efficiently taint and untaint instructions. The taint tracking mechanism, proposed by SPT, controls both non-speculative and speculative execution and is able to protect secrets that reside in registers before speculation. Also, the taint tracking mechanism enables SPT to leverage their key insight: "*any data that can leak through the program's non-speculative execution should not be treated as secret during the program's speculative execution".* While SPT can maintain taints in both non-speculative and speculative execution, it can dynamically untaint and thus execute instructions normally under speculative execution. Whenever SPT identifies leaked data, its untainting mechanism untaints both older and younger dependent tainted instructions. SPT protects against the leakage of all non-speculative secrets — addresses that have never leaked their values non-speculatively.

3 THREAT MODEL

In the following section, we outline ReCon's threat model, specifically how it integrates with the STT and NDA threat models, and the modifications ReCon makes to the visibility of potential secrets.

3.1 STT and NDA Integration

ReCon is a performance optimization that is applied on top of secure speculation mechanisms, e.g., NDA or STT. The underlying secure mechanism maintains its entire threat model *except for values that previously have been made public and, hence, not guaranteed to be protected under speculation by the secure mechanism.* More specifically: In both NDA and STT secrets are defined as values that the core should not be able to access, and is only able to access as a result of erroneous speculation, i.e., potential secrets are speculatively accessed values that might be squashed. ReCon, similarly to SPT [14], relaxes this definition by excluding values that have already been made public through non-speculative execution.

Regarding the threat models of STT and NDA with ReCon we note the following:

Register Protection: For both STT and NDA, values that reside in registers before the point of speculation are not considered to be potential secrets, as these are accessible without speculation, and are therefore part of normal execution. ReCon does not affect this property, i.e., does not add protection for registers.

Explicit and Implicit Channel Protection: STT delays potential transmitters to defend against explicit channels. For implicit channel mitigation, STT ensures that the control flow is not influenced by tainted data. NDA delays the propagation of a speculative load

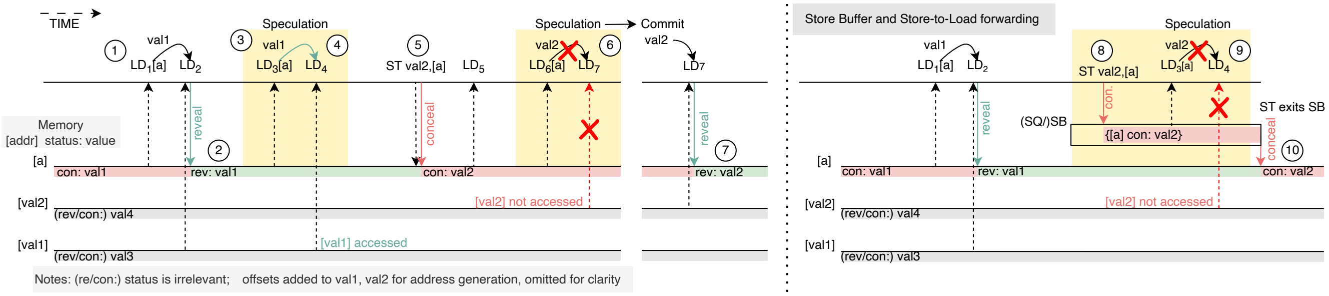

Figure 1: Execution overview of ReCon.

instruction, and thus it does not need special treatment for explicit and implicit channels, as they cannot be formed. ReCon does not affect either of them (for secrets that have not been made public).

Speculative Model: STT introduces two speculative models, *Spectre* that considers only speculation cast by control flow instructions, and *Futuristic* that considers an instruction speculative until it guarantees not to be squashed. ReCon can operate under any speculative model, and it is not affected by the model choice. For evaluation, we use only speculation cast by control flow and store instructions, we elaborate more on that in Section 6.

Lastly, we do not examine potential issues stemming from simultaneous multi-threading (SMT) [50], as such features are typically disabled in secure systems [33, 46], or not even supported (e.g., Apple M1 [18, 51]).

3.2 Conditional Security Guarantee

As described, ReCon removes protection from values that are already public information. The lack of protection for public information enables some new forms of execution patterns that were not previously possible in the underlying threat model. For example, load-load pairs that have previously been observed non-speculatively would be allowed to execute speculatively, as the information is considered public under the threat model. Programs that utilize secret-dependent non-speculative behavior are considered to have made their secrets public, even if such accesses were attempted to be obfuscated. Take the following example:

```
1: for (i = 0; i < N; i++) {
2:   tmp = AES_KEYS[i] // Obfuscate access
3: }
4: selector = key_selector[iteration] // Find key
5: key = AES_KEYS[selector] // Select key
```

The secret value, `selector`, is made public at line 5, in which observable program behavior is dependent on the selector value. This means that the threat model does not consider future leakage of this value as insecure, since it has been made public. Thus, it would be possible to create speculative replay attacks to expose the value of `selector`, leveraging a speculative gadget (similar to lines 4 and 5 in the example above) elsewhere in the code. The example attempts to obfuscate which key is selected through accessing all possible keys, but such methods are not considered secure in general.

Speaking generally, this means that programs that avoid secret-dependent behavior, as recommended by modern secure programming techniques, in their non-speculative execution will not have their security premise changed under the ReCon threat model. Observe a secure version of the previous example:

```
selector = key_selector[iteration]
key = 0
for (i = 0; i < N; i++) {
  tmp = AES_KEYS[i] // Access all keys
  // Constant time selection of key
  // cmp=1 if i==selector else cmp=0
  cmp = (1 - min(selector - i, 1))
  mask = cmp * 0xFFFFFFFF
  key |= mask & tmp
}
```

This version employs constant-time programming principles that ensure that the secret, `selector`, is never part of observable program execution. Such programming principles are endorsed by industry leaders such as Intel [20] for secure applications.

Sandboxing: ReCon does not inherently support isolation between multiple sandboxes in the same process (address space). For example, consider two sandboxes operating in the same process. As public information can be accessed speculatively with ReCon, one sandbox can access and speculatively “observe” public information of the other sandbox. This “observation” would normally be prevented by NDA or STT. Today, this is of less practical concern, as sandboxes are separated in their own process, e.g., in web browsers (Site Isolation) [34] or in the cloud [42].

4 RECON

In the following section, we describe the core premise of ReCon. We describe i) how we use non-speculative leakage to improve the performance of secure speculation schemes, ii) how we can detect and capture this non-speculative leakage (*reveal*), and iii) how we can ensure that new secrets are not accidentally leaked (*conceal*).

4.1 Overview

ReCon uses knowledge of prior non-speculative execution to alter the execution of specific load instructions under speculation. The goal is to lift speculative side-channel defenses for loaded values that are safe to improve performance.

ReCon identifies pairs of dependent loads, in which the second address is entirely dependent on the output value of the first load (see ① in Figure 1), and marks this value as revealed when the second load commits ②. For now, assume that a bit associated with each word in the cache hierarchy marks the word as revealed or concealed. We discuss the details of the storage and transmission of this information in Section 5.

This means that the *value* of the first load has leaked (non-speculatively) as an address via a side-channel and should not be considered a secret as defined by the SPT threat model [14]. Protecting this value under speculation is superfluous for security, but harmful to performance.

ReCon essentially marks the address accessed by the first load ($\mathtt{LD}_1$) as containing a revealed value, and leverages this knowledge to disable security mechanisms for any load to this address ③, as these security mechanisms are detrimental to performance. The value is only safe as long as it has not been changed, i.e., ReCon marks the address of stores as concealed ⑤.

ReCon tracks *direct dependence* load pairs (Section 4.3 and Section 5.1), preserving the address of the first load ($\mathtt{LD}_1$) until the non-speculative commit of the second load ($\mathtt{LD}_2$), and then marking the corresponding word in the cache as revealed. Whenever a load is performed to a revealed value, special handling appropriate to the underlying security mechanism ensures that the revealed value is treated as though it is non-speculative ④, e.g., it does not cause taints for STT, and can propagate immediately for NDA.

4.2 What Non-Speculative Leakage to Capture?

Non-speculative information leakage occurs as a result of changes to the microarchitectural state that are visible to an attacker, such as timing differences in execution. When disregarding approaches that focus on energy usage or contention, microarchitectural changes that leak information can only occur in two possible ways:

(1) through a data dependency between a load and a following transmitter instruction, known as an explicit channel.
(2) through a control dependency where the different possible paths result in different microarchitectural states, known as an implicit channel.

ReCon efficiently captures non-speculative leakage to improve the performance of an underlying secure speculation scheme. For such a case, leakage from explicit channels is of the greatest interest, because most of the performance loss in secure speculation schemes comes from the reduction in memory-level parallelism, as a consequence of preventing explicit channels [56]. Leakage from control dependencies is harder to detect and causes less performance degradation under secure speculation schemes [56].

Within explicit channel leakage, we focus solely on leakage caused by dependent loads (load pairs), which is common through pointer dereferencing. As we analyze in Section 6.2, this case constitutes the majority of leakage caused by load instructions. Although this leaves a subset of leakage undetected, this is not a correctness issue for security; it only affects the attainable benefit we can achieve. Comprehensive protection is provided by the underlying secure speculation scheme that we are optimizing.

4.3 Direct Dependence Loads

Non-speculative leakage due to data dependencies can occur, e.g., when a load passes its value directly to another load, or when a load gets its value manipulated by a sequence of instructions that passes it to a subsequent load. Consider the following example:

```
PC1: load   r1, [0x13]
PC2: load   r2, [0x7]
PC3: add    r3, r1, r2
PC4: load   r4, [r3]
PC5: load   r5, [r1]
```

Values are loaded (PC1 & PC2) from two addresses (0x13 and 0x7), manipulated (PC3), and passed as an address argument to a third load (PC4). PC4 is a transmitter instruction that leaks the addresses 0x13 and 0x7 through *indirect dependencies*. The value from address 0x13 (loaded by PC1) is also passed as an address argument to a fourth load (PC5). PC5 is also a transmitter instruction that leaks address 0x13 through a *direct dependence* (meaning no other instruction intervenes between the two loads). In this example, the value stored in 0x13 is leaked both indirectly (through a dependence tree that involve other instructions and, importantly, may involve other loads) and directly (through a load-load pair without any dependent intervening instructions). ReCon limits its detection to only the leakage of load pairs with a *direct dependence*.

ReCon associates a leaked value with the address it is stored at (e.g., 0x13 in the example). Establishing this association for an arbitrary large indirect-dependence tree requires dynamic information flow tracking [45] that extends arbitrarily long in the past and may involve multiple loads and addresses (e.g., PC1 *and* PC2, 0x13 *and* 0x7, in the example above). In contrast, with exactly two loads (PC1 *and* PC5) it is straightforward to unambiguously associate the leakage to the address (0x13) in a simple and effective way.

Load addresses that are derived from a load with an offset, i.e., an immediate, still create a valid load-pair, as the introduction of an offset does not affect ReCon's overall security guarantee. Such an offset is by definition always present, and calculating the value of the secret is trivial. Consider the following example:

```
PC1: li   r1, 0x13
PC2: load r2, [r1 + 0x10] // 0x10 is an offset
PC3: load r3, [r2]
```

Instead of revealing address 0x13, it reveals address 0x13+0x10. If instead, the second load has an offset:

```
PC1: li   r1, 0x13
PC2: load r2, [r1]
PC3: load r3, [r2 + 0x10] // 0x10 is an offset
```

The load pair now reveals address 0x13, as the offset is known.

ReCon is secure regardless of the proportion of total non-speculative information leakage it captures, as security is guaranteed by the underlying secure speculative execution scheme, and ReCon is only using already leaked information to reduce their performance overhead. As such, ReCon has a trade-off between the design cost of capturing information leakage, and the performance gain from using this leakage. ReCon uses this trade-off to capture most leakage at a low cost by focusing on loads whose address is directly dependent on another load without intermediaries, which comprises a majority of the total leakage of a program (Section 6.2). Detecting the presence of such direct dependence load pairs is achieved in a simple manner by checking to see if there is a dependence between the destination register of a preceding first load and the source register of a following second load.

We describe such an implementation in detail in Section 5.1. Then, once the second load is committed, the leakage of the value is known to be non-speculative, and the address of the value is marked as revealed in the cache, which enables following load instructions to that address to lift security mechanisms.

Insofar as CISC instructions are internally decoded into RISC micro-operations, instructions such as the x86 arithmetic instructions with a memory-fetched input operand are broken into two (or more) micro-operations where one of them is a load. This load can participate in the formation of a load pair.

4.4 Concealing New Secrets

Once an address is revealed, it is guaranteed to be safe as long as its contents do not change. A store to an address that has been revealed breaks this condition, making the address unsafe again. In this section, we describe how ReCon assures that revealed addresses turn to concealed when they are changed. More importantly, conceal operations work at *any granularity* (e.g., byte, sub-word, word, ...), while revealing and using revealed data works only for aligned addresses and at word granularity. This means that if any part of a revealed word changes, the whole word becomes concealed.

4.4.1 Performed Stores. When a committed store instruction writes to its target address in the cache hierarchy (i.e., when the store is *performed*), the new contents at this address have not been observed non-speculatively (i.e., revealed through a committed load instruction) (⑤ in Figure 1). For this reason, a store marks the address as *concealed*, (no longer revealed) — the address contains a new secret — which prevents future speculative loads from passing the loaded value as an address to other instructions ⑥. Upon commit of a dependent load that uses the contents of the concealed address, the address is marked anew as *revealed* ⑦.

4.4.2 In-Flight Stores. We consider committed stores that reside in the store buffer (SB) as not yet performed; stores in the store queue (SQ), are in-flight stores that have not committed. Younger loads receive their value forwarded from stores in the SB or in the SQ, rather than from the L1 cache.

In ReCon, a store conceals its output in the SQ/SB (⑧ in Figure 1). Thus, a load always receives concealed data from store forwarding and defenses cannot be lifted ⑨. There may be a period where the same data are known as *concealed* inside the core and *revealed* outside. This is inline with memory models that relax the *store* → *load* order and are *read-own-write-early multi-copy atomic (rMCA)* [49], e.g., x86-TSO [43] or weaker memory models. The memory location is *concealed* outside the core when the store exits the SB ⑩.

4.5 Store-to-Load Forwarding

Store-to-load forwarding forms an implicit branch that can potentially leak speculatively accessed secrets. The following section describes how we ensure that ReCon does not leak potential secrets through such implicit branches.

```
if (r1 < size) {
  // Access
  PC1:  load  r2, [r1]
  PC2:  store r3, [r2]
  // [r4] is revealed
  // Transmit
  PC3:  load  r5, [r4]
  PC4:  load  r6, [r5]
}
```

(a) Store-to-load forwarding

```
if (r1 < size) {
PC1:  load  r2, [r1]
PC2:  store r3, [r2]
PC3:{ // Load behavior
   impl-if {r2 == r4)
      // Concealed
      r5 = r3
   impl-else
      load  r5, [r4]
   }
PC4:{ // Load behavior
   impl-if {r2 == r5)
      // Concealed
      r6 = r3
   impl-else
      load  r6, [r5]
   }
}
```

(b) Implicit channels

Figure 2: Implicit channels of store-to-load forwarding with ReCon.

4.5.1 Without Memory Dependence Speculation. An unresolved store forms a *resolution-based* implicit channel that is handled by delaying loads until the store address is non-speculatively resolved [56]. In this case, ReCon has no effect.

4.5.2 With Memory Dependence Speculation. For a pair of dependent loads (PC3 and PC4), two implicit channels are formed in the presence of an older unresolved store (PC2), as shown in Figure 2. With memory dependence speculation [15, 31], two memory dependence predictions take place. The implicit channels also become prediction-based channels that are mitigated by updating the predictor with non-speculative values [56].

As outlined in Table 1, there are four cases to consider: Either of the two loads can be predicted to depend on the previous store (STF) or on memory (MEM). The key takeaway is that what is observable under STT and what is observable under ReCon differs only in the first case, (assuming that address [r4] is *revealed* — otherwise there is no difference). This is expected as if ld [r4] (PC3) is observed and [r4] is *revealed*, ReCon allows ld [r5] (PC4) to also be observed. This does not leak anything more than what has leaked before. In case 2, the store forwarding passes a *concealed* value to ld [r5] (PC4) preventing it from being observed both in STT and ReCon. Similarly, for case 3 and 4, the store passes a *concealed* value to ld [r4] (PC3, effectively reverting ReCon to STT. Thus, in both STT and ReCon, the observable loads are independent of the speculatively loaded secret (PC1), and the only information that leaks, is the memory dependence predictions, which are independent of the secret. A similar argument holds for NDA permissive propagation [52].

Table 1: Memory dependence prediction cases for the store forwarding example of Figure 2.

	Prediction		STT	ReCon
Case:	PC3	PC4	observation	observation
1	MEM	MEM	ld [r4], —	ld [r4], ld [r5]
2	MEM	STF	ld [r4], —	ld [r4], —
3	STF	MEM	—, —	—, —
4	STF	STF	—, —	—, —

5 IMPLEMENTATION

In this section, we describe how ReCon can be implemented. We first describe how load pairs are detected, how revealed addresses are tracked in the cache hierarchy, and finally how revealed addresses are exploited to lift security defenses for NDA and STT.

5.1 Detecting Non-Speculative Load Pairs

An address that is used by a direct dependence load pair is only safe once the second load (LD$_2$ in Figure 1) becomes non-speculative. From a strict security perspective, the earliest that LD$_2$ becomes non-speculative is when the load has reached its visibility point [56], i.e., it is bound to commit. Although a load can reach its visibility point anywhere in the pipeline, for simplicity, we opt for implementing load-pair detection in the commit stage. This has no impact on security, as it only delays the revealing of the address and the earliest point when security defenses can be lifted.

ReCon detects load pairs by including a table with, at most, as many entries as physical integer registers (smaller tables are possible and are evaluated in Section 6.6). Each table entry consists of an active (A) bit and an address field. We call this table the load-pair table (LPT), and it is accessed using the indices of the *source* and *destination* physical registers of a committing instruction. Detecting load pairs at commit, using physical registers, relieves us from the burden of establishing the correct dependence when multiple dynamic instances of the same load pair exist in the pipeline.

When a load commits, the LPT is accessed using both the destination register and the source register. For the destination register, if the load address has not already been revealed, then the active bit is set and the load's address is written to the corresponding entry, see ① in Figure 3. At the same time, the load checks the active bit of its source address register ②. If the active bit is set, then a load pair has been detected and the load is the second load of the pair (LD$_2$). The committing load, then, marks the address that is stored in the LPT entry (i.e., of the first load) as revealed. Revealed addresses are tracked in the cache hierarchy, as described in the next section. If the active bit is not set, then no further operations are performed. The active bit is cleared for the destination register(s) of any other instruction than loads that commit.

5.1.1 Multi-Source Load Instructions. Up to this point, we have described how ReCon tracks load pairs where the second load has a single direct dependence on an older load, i.e., the second load having a single source register. Let us now consider more complex instructions that might have multiple source registers, as commonly found in the x86 instruction set.

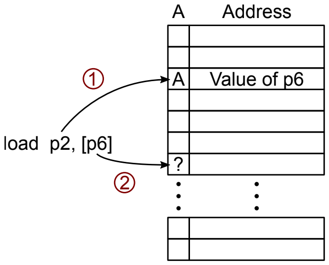

Figure 3: The load pair table. p2 and p6 represent the renamed physical registers, 'A' is the active bit, and the 'Address' field holds the address accessed by a committed load.

Consider the following x86 assembly, which is the result of compiling the commented out code:

```
# selector = key_selector[iteration]
# output = AES_KEYS[iterator];
mov   0x0(,%rax,8),%rax     # %rax = key_selector
mov   (%rdx,%rax,8),%rax    # %rdx = AES_KEYS
```

At the first sight, one might conclude that the two instructions form a load pair, as the second mov instruction has a direct dependence on the first mov through register %rax. However, the actual behavior depends on the underlying microarchitecture and its micro-operations. Such complex instructions are commonly decoded into multiple simpler micro-operations. For the given example, the second mov could be decoded into three micro-operations, as follows:

```
# mov   (%rdx,%rax,8),%rax
mul   %rax,8,%r1            # src1,src2,dst
add   %rdx,%r1,%r2          # src1,src2,dst
load  (%r2),%r3             # src,dst
```

In this case, ReCon would not detect a load pair since there is no direct dependence between two load operations. If the underlying microarchitecture instead support multi-source operations, such that the second mov is decoded into a single micro-operation, then a load-pair would be detected.

In the general case, the second load micro-operation can have as many direct load dependencies as it has source operands, and a load pair can be detected for each operand. This requires that a lookup is made in the load pair table for each source operand to detect if the source register is written by a load operation. Each load pair can then be revealed by sending a reveal message to the L1. This is not on the critical path and such messages can be dropped since not revealing an address is only a potential performance loss and is always secure to do. For this paper, we focus on evaluating the case where there exists only a single direct load dependence and leave multi-source operations for future work.

5.2 Tracking Revealed/Concealed Addresses

ReCon tracks revealed addresses by associating a bit vector with each cache line to mark the data words that have been revealed by committed loads. The vector has as many bits as the total number of words that fit in a cache line (e.g., eight bits for eight 8-byte words in a 64-byte cache line—see Section 6.7 for details).

A newly fetched cache line from memory has all its words marked as concealed. When the second load (LD_2) of a load-pair commits, the address accessed by the first load (LD_1) is marked as revealed. LD_2 sends a `reveal` request to the L1 cache to reveal the address by setting the bit of the corresponding word in the cache line. Similarly, upon the commit of a store instruction, the bit of the corresponding word is cleared, stating that a new value exists in the data word that has not yet been revealed.

5.3 ReCon Coherence

A similar bit-vector approach to maintain reveal/conceal information per address, is followed in SPT to keep taints/untaints, but instead of keeping the bit-vector with every cache line in the L1, a different structure, mirroring the L1, keeps the bit-vectors separately [14]. While the SPT approach has the advantage of i) bounding the absolute storage cost, and ii) not changing the L1, it has two important disadvantages when it comes to using non-speculative leakage information for optimization: First, leakage information is *private* to each core and *cannot be shared*. Today, several important workloads are multithreaded (e.g., browsers) to be able to extract performance out of multicores. In ReCon, we want to take advantage of the information gained in one core to optimize the execution of another core as the security model applies for a whole process (subject to the restrictions discussed in Section 3). Second, naïvely propagating this information to a shared LLC, e.g., via evictions, is not coherent and may result in a loss of information.

A contribution of ReCon is to solve this problem by assigning the non-speculative leakage information as meta-information that is carried and maintained by the coherence protocol. For this work, we assume a standard directory MESI protocol. A coherent version of the ReCon bit-vector is kept with each directory entry. The bit-vector of a cache line is transferred with the standard coherent transactions of the protocol between the directory and the private caches and between the private caches themselves.

Consider a single cache line, shared by threads of the same process (same address space) running on two different cores with private L1 caches. Each L1 receives a copy of the directory bit-vector, initially all set to concealed. Each thread can independently reveal words in this cache line in its L1 bit-vector, without knowledge of the other thread's bit-vector state. At this point, the revealed information can only be used locally by each core. However, upon eviction, the directory needs to be notified that the particular L1 is no longer a sharer (for this cache line). It is at this point that the ReCon bit-vector is transferred back to the directory and is logically Or'ed with the bit-vector that exists there. Or-ing the L1 bit-vector with the directory bit-vector guarantees that information is preserved across consecutive evictions from different L1s. Any core that reads the cache line from the directory, now learns of all the revealed addresses accumulated in the directory bit-vector.

Consider, now, what happens when a core *conceals* an address. Recall that to conceal an address, its contents must change, i.e., the address must be written by a store. For this, the L1 needs to have write permissions to its cache line. If the cache line is not already in state `M` (`Modified`), the core must ask the directory to grant write permission and invalidate any other sharers.[1] In ReCon, the writer assumes control of the directory bit-vector and owns the only coherent copy of the ReCon bit-vector until either: i) writes it back to the directory (*overwriting* the directory bit-vector with its own copy); ii) writes it back to the directory and passes it on to a new reader on a downgrade; or, iii) passes it on to the next writer on an invalidation (from the new writer). Until the writer gives up its write permission: i) it can reveal as many addresses in the cache line as it wants (which no other core can do at the same time); and ii) it is the only core that can supply a valid bit-vector to (a request from) a new reader or a new writer.

5.4 Using Revealed Addresses

Load instructions that perform a cache access, eventually return the corresponding ReCon bit in the cache hierarchy for that word. If the value has already been revealed, then the core can immediately disable any applied speculative restrictions associated with the loaded value. For NDA [52], the loaded value of revealed addresses is immediately propagated to dependent instructions. Similarly, for STT [56], any load that receives a revealed value untaints its destination register, which enables the value to be used by any transmitting instruction. Both techniques benefit from increased instruction and memory-level parallelism.

6 EVALUATION

In this section, we describe our methodology, we characterize the non-speculative leakage, and we present ReCon's overall results.

6.1 Methodology

We implement the evaluated security schemes on the latest version of the gem5 [12] simulator (version 22) using Ruby and SLICC to model the memory system with a three-level MESI coherence protocol (with an in-cache directory), on an infrastructure that shares implementations for NDA, STT, and ReCon. We use GARNET [1] to model the interconnect.

Speculation state is tracked through shadows [38, 39]. We evaluate only speculation that is triggered by control and store instructions [52], similarly to other speculative side-channel threat models [14, 53, 55, 56], which only track control instructions, as this is the type of speculation leveraged by Spectre attacks [10, 22, 24, 41]. We do not consider speculation triggered by load instructions and instructions that can raise an exception for the following reasons: there is not a discovered attack under speculation caused by load instructions, and recovering the performance lost from this type of speculation is a solved problem [36, 48, 57], and speculation caused by instructions that can raise an exception lie on a different spectrum of attacks [29]. Thus, we are closer to the Spectre threat model that tracks only control shadows [56], rather than the Futuristic threat model that considers all instructions speculative until they reach their visibility point [56].

We use the SPEC2017 speed [17] and SPEC2006 [16] CPU benchmark suites as a representative for single thread applications, and

[1] The private ReCon bit-vector of an invalidated *reader* is lost in the invalidation. A potential optimization would be to try to preserve it, but we have omitted this for simplicity.

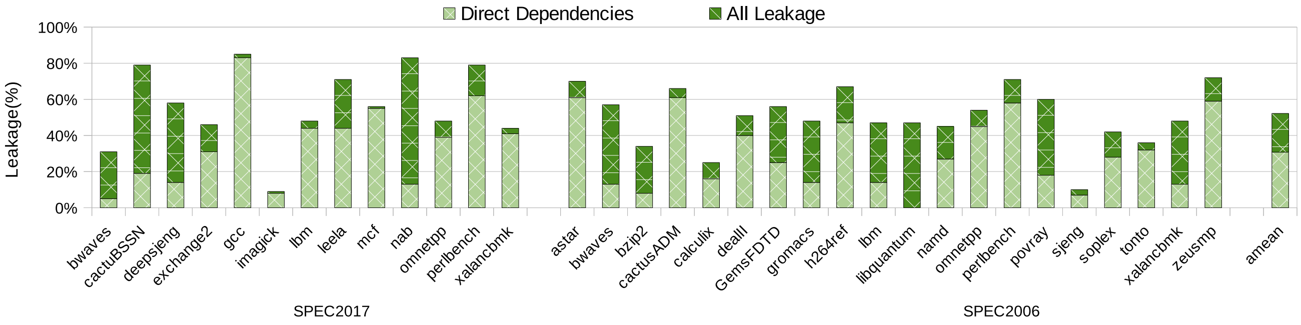

Figure 4: Percentage breakdown of leakage out of all address space.

Table 2: gem5 Configuration

Processor	
Core	3GHz (4-cores for parallel benchmarks)
Decode width	8 instructions
Issue / Commit width	8 instructions
Instruction queue	160 entries
Reorder buffer	352 entries
Load queue	128 entries
Store queue/buffer	72 entries
Memory	
L1 D & I caches	64KiB & 64KiB, 8 ways 2 cycles roundtrip data
L2 cache	2 MiB, 16 ways 6 cycles roundtrip
LLC cache	16 MiB, 32 ways (4-cores: 4 MiB each) 16 cycles roundtrip
Coherence Protocol	3-level MESI
Coherence Directory	In-Cache (LLC)
Cache line Size	64 bytes
Memory	8 GiB

use the PARSEC [11] benchmark suite to evaluate parallel benchmarks. For the SPEC2017 benchmarks, we run detailed full system (FS) simulations using the out-of-order (OoO) processor model, extracting simulation phases with the use of simpoints [44]. We collect simpoints for the first 100 billion instructions (under FS simulation). We take up to five simpoints per benchmark for an interval of 100 million instructions. We start gathering statistics after running 50 million instructions of a detailed warm-up, so that our mechanism that is implemented in OoO is also included in the warm-up. For the SPEC2006 benchmarks, we run in system emulation (SE) mode using the out-of-order (OoO) processor model. We warm up the processor for 3 billion instructions, and we gather statistics for the next 1 billion instructions. For the parallel benchmarks, we fast-forward into the region of interest (ROI) in system emulation (SE) and run for 100 million instructions. We present instructions per cycle (IPC) as a performance metric for SPEC2017 and SPEC2006, and ROI execution time for PARSEC [6].

We also use Clueless [13], an open-source tool that measures the exposure of a program's memory to cache side-channels by applying a global DIFT mechanism, to better understand the general behavior of non-speculative leakage. Clueless is a trace-based tool that does not model speculative execution, and thus models the non-speculative behavior of the program. Clueless tracks dynamic instruction dependencies (through registers and memory) and detects data values that are turned into addresses. These are considered leaked values and their address in memory is tagged as a leakage point. Newly written values revert the address back to a non-leaked status. Thus, Clueless dynamically records the portion of memory that has leaked at any specific moment.

Because non-speculative leakage due to direct-dependence load pairs (Section 4.3) is a subset of the leakage captured by DIFT, we modify Clueless to also provide statistics specifically for direct dependence load-to-load dereferencing. We also modify it from being pin-based to trace-based, and we use SPEC2017 and SPEC2006 traces provided by the ChampSim [19] simulator for general studies.

6.2 Leakage Breakdown

To understand the direct-dependence load-to-load leakage, we report results from Clueless. In Figure 4, we show the average percentage of memory addresses that are identified as leakage points. We show the results both for all captured leakage (DIFT) and for direct dependencies (load-load pairs). We observe that across the SPEC2017 and SPEC2006 benchmark suites, on average, 53% of the address space leaks its content (when we capture leakage by DIFT), while direct load-to-load dereferencing is responsible for 32% of the address space that has leaked its content. In other words, direct dependencies cover 60% of the total leakage. In fact, we find that in some cases there is negligible additional leakage occurring if we measure it with DIFT, and *the program's leakage is solely due to direct-dependence load-to-load dereferencing* (e.g., `gcc`, `imagick`, `mcf`, and `xalancbmk` from SPEC2017).

6.3 Performance Results

Figure 5 and Figure 6 show the performance results of NDA/STT and ReCon for single thread performance as instructions per cycle (IPC) normalized to the unsafe baseline processor.

The more strict NDA introduces a 13.2% performance degradation on average, while STT introduces a degradation of 8.9% compared to the unsafe baseline across the SPEC2017 benchmarks.

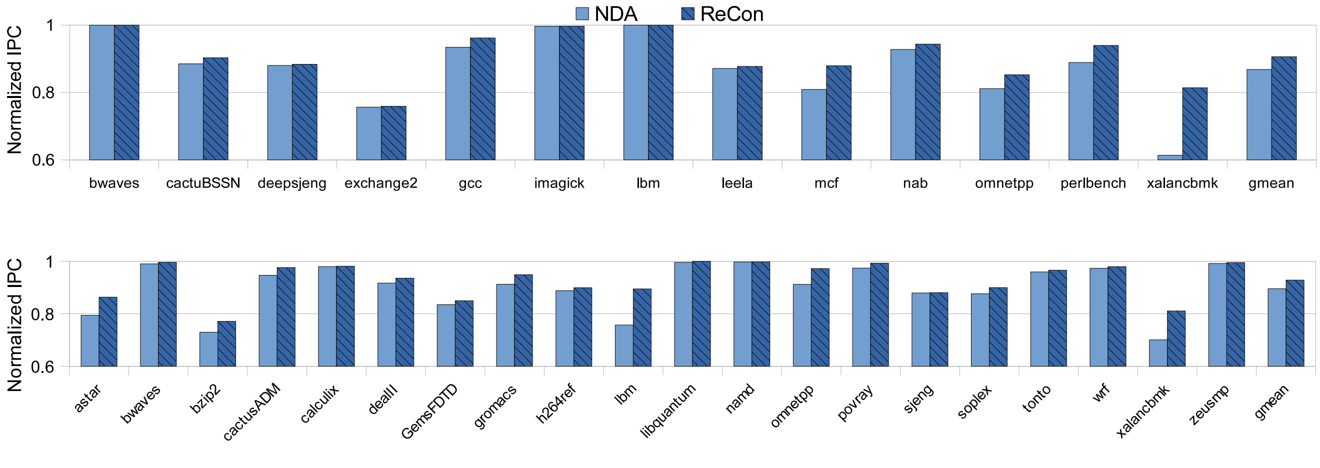

Figure 5: Normalized IPC to unsafe baseline of NDA+ReCon, on SPEC2017 (upper) and SPEC2006 (lower).

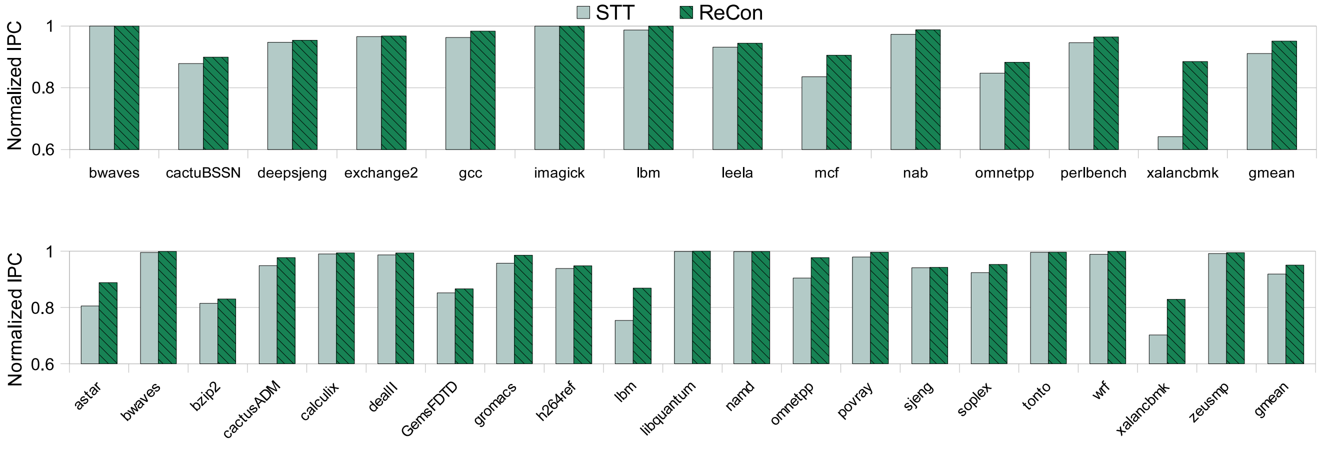

Figure 6: Normalized IPC to unsafe baseline of STT+ReCon, on SPEC2017 (upper) and SPEC2006 (lower).

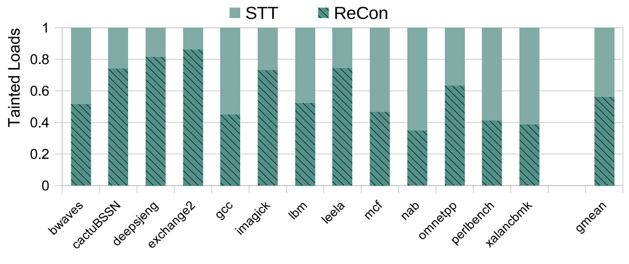

Figure 7: Amount of tainted loads on SPEC2017 of STT (full column) and ReCon (hatched part), normalized to STT.

For the SPEC2006 benchmark suite, NDA introduces an overhead of 10.4%, while STT introduces an overhead of 8.1%, on average.

ReCon's purpose is to reduce the number of tainted (STT) or not-propagated (NDA) load instructions, and thus increase performance. Figure 7 shows the reduction in tainted load instructions for ReCon normalized against the total number of tainted loads for STT. We see significantly fewer, 43.8% on average, tainted loads with ReCon. This is a natural consequence of the mechanism, as ReCon untaints the destination register, and thus does not cause dependent loads to be tainted. ReCon's improvement for NDA are nearly identical, since both STT and NDA apply their defenses to the same loads (i.e., loads depending on speculatively loaded values), and ReCon applies its optimization to the same set of loads. We have therefore omitted the data for NDA to simplify the figure.

For SPEC2017, ReCon faces a performance overhead of 9.4% and 4.9% over the unsafe baseline processor, reducing it by 28.7% for NDA and 45.1% for STT, respectively. For SPEC2006, we observe similar results, with an overhead of 7.2% and 5%, which translates to a reduction of 31.5% and 39% for NDA and STT, respectively.

Notice that some benchmarks face a very small absolute number of tainted loads (i.e., `bwaves`, `imagick`, and `lbm` from SPEC2017)

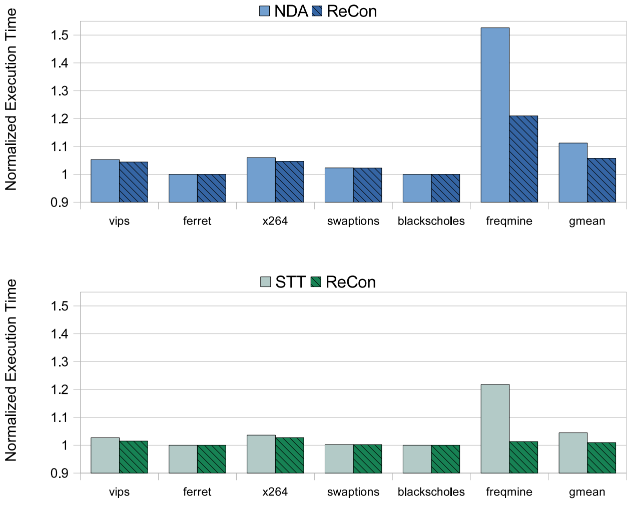

Figure 8: Normalized execution time of parallel benchmarks.

and they do not face a performance degradation on either NDA or STT (Figure 5 and Figure 6), leaving no room for ReCon to boost the performance. Another notable mention is that some benchmarks have a similar absolute number of tainted loads (i.e., `leela` and `nab`), yet the former faces a performance reduction of 6.8% while the latter only 2.7% (on STT). This observation shows that some tainted loads are more critical than others, and reducing the number of tainted loads does not guarantee analogous performance gains. For example, ReCon recovers a significant amount of performance (from 64.1% to 88.5%) by reducing the number of tainted loads by 61% for `xalancbmk` (SPEC2017) as compared to STT, while for `perlbench` (SPEC2017), ReCon improves the performance from 94.6% to 96.4% by reducing a similar amount of tainted loads (59%). The former reduces the overhead by 67.9% while the latter by 34.5%, yet the reduction in tainted loads is similar (with the latter having slightly more reduction). This can be seen in Figure 7.

For the PARSEC benchmark suite, NDA increases the total execution time by 9.7% and STT by 4.4%, as shown in Figure 8. ReCon reduces the execution time overhead by 46.7% and 78.6%, resulting in a slowdown of 5.2% and 1% over the unsafe baseline, respectively.

6.4 Leakage/Performance Correlation

To understand how detected leakage from load pairs correlates with the performance gains of ReCon, we analyze benchmarks that experience at least a 5% performance degradation with STT for SPEC2017, namely `cactuBSSN`, `deepsjeng`, `mcf`, `leela`, `omnetpp`, `perlbench`, and `xalancbmk`. The 5% performance degradation limit reduces noise from benchmarks where STT and ReCon marginally affect performance.

Figure 9 illustrates the correlation between non-speculative leakage (as observed by Clueless) and performance (as observed by ReCon). The figure shows the ratio of leakage captured by direct-dependence *load pairs* to *all leakage* captured by Clueless' global DIFT mechanism. A perfect ratio means that all leakage is captured by load pairs and would be represented by a full column. Benchmarks are sorted from higher overhead reduction to lower overhead reduction (left to right). We see that ReCon successfully recovers performance when the leakage is highly dependent on load pairs. The lower the ratio of load-pairs to total leakage, the lower the performance gain (e.g., `cactuBSSN` and `deepsjeng`). Moreover, the amount of performance gains is dependent on two things: i) the rate at which pointers are reused (i.e., previously seen pointer dereferencing should be repeated), and ii) the phase the pointers are reused: ReCon requires the program to experience speculative execution when reusing them (and thus the underlying secure speculation scheme is applied).

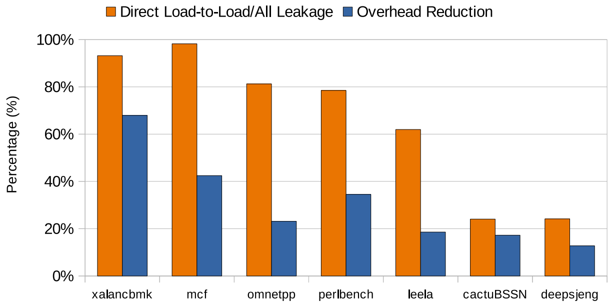

Figure 9: Correlation between percentage of captured leakage (direct load pairs / all leakage) and overhead reduction. (SPEC2017 benchmarks with more than 5% performance degradation shown.)

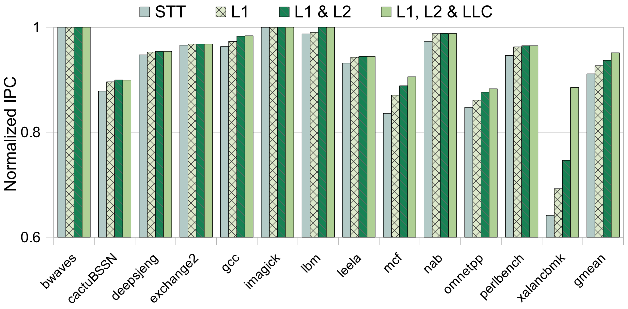

Figure 10: Normalized IPC comparison of ReCon when applied to different cache levels.

6.5 L1 and L2 bound ReCon

ReCon is a flexible optimization that can be applied to multiple cache levels. While the default design applies the mechanism to all levels (L1, L2, and LLC), we examine the behavior when it is applied to only the first cache level, and on the first and second cache level, thus introducing a low implementation overhead.

Figure 10 shows the evaluation for STT with the SPEC2017 benchmark suite. We observe that some benchmarks, such as `cactuBSSN` and `leela`, recover the majority of the performance loss only by using the L1 cache, while others, such as `gcc`, `mcf`, `omnetpp` and

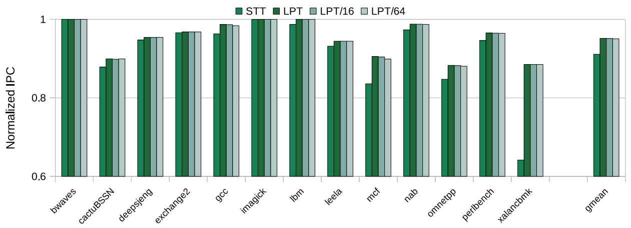

Figure 11: Normalized IPC of STT+ReCon with various sizes of Load-Pair Table (LPT).

xalancbmk, need to cover a larger working set size and thus leverage the L2 and LLC.

Overall, applying ReCon only to the L1 data cache reduces the overhead introduced by STT from 8.9% to 7.3%, and applying ReCon to the L1 and L2 reduces the overhead further to 6.3%.

6.6 Load-Pair Table Sensitivity Analysis

The load-pair table (LPT), as described in Section 5.1, uses as many entries as physical registers to store the address accessed by a load instruction. While the number of registers is architecture-specific, modern architectures commonly have around 200 integer registers. More specifically, Intel Skylake has 180 integer registers [21] and AMD Zen 3 and Zen 4 has 192 [7] and 224 [8] integer register, respectively. For that many registers, the LPT would translate to a size slightly bigger than 1KiB (we elaborate more on the implementation overhead in Section 6.7). As direct-dependence load pairs are usually near each other (in program order and inside the pipeline), just a few entries are enough to capture the majority of load pairs.

Figure 11 shows the results of a sensitivity study where we successively reduce the LPT size by a factor of two. The table is still indexed by the destination and source registers, but now conflicts are possible, as different physical registers map to the same entry. To ensure correctness, we tag LPT entries with the physical register index. The results show that the only benchmark that is significantly affected by the LPT size, experiencing increasingly many conflicts with every reduction, is *mcf*. We evaluate all the configurations between LPT/2 and LPT/64. They are consistent with the trend shown in the figure, thus, we omit them to simplify the presentation.

Overall, we observe that reducing the LPT size marginally affects performance. This behavior verifies our assumption that load pairs consist of loads that are close to each other.

6.7 Implementation Overhead

ReCon is a low-complexity approach, and its implementation overhead is primarily a storage overhead in the cache hierarchy and directory. More specifically, ReCon makes changes in the core and the cache hierarchy as follows.

In the core, ReCon adds a load-pair table in the commit stage to propagate the accessed address of committed loads. The load-pair table (LPT) consists of an address (48 bits) and a valid bit (1 bit) per entry. For example, 180 registers (Intel Skylake [21]) would require a 1.1KiB LPT, while 224 registers (AMD Zen 4 [8]) would require a 1.37KiB LPT. As explained in Section 6.6 this could be further reduced to 641 bytes and 798 bytes, respectively, by shrinking it to half and adding an extra eight bits per entry (tag for register).

ReCon works with aligned 8-byte memory locations, to limit the total number of reveal bits required per cache line. It does not do misaligned or sub-8-byte *reveal* operations and keeps the values concealed in such cases. In the cache hierarchy, ReCon adds a byte per 64-byte cache line in the private caches and in the directory to track the revealed/concealed state of memory locations (eight revealed memory locations can fit at maximum in a 64-byte cache line). We evaluate an in-cache directory, which makes the storage cost of the directory bit-vectors proportional to the LLC size. This translates to an overhead of less than 1.5% of the total cache storage (private caches and LLC, considering the storage cost of data + tags + coherence state). For a high performance system, one can consider a decoupled directory that is, e.g., 2× or 4× over-provisioned compared to the aggregate size of the private caches. In that case, the storage cost of the directory bit-vectors becomes proportional to the aggregate size of the private caches.

7 RELATED WORK

Several approaches have been proposed to defend against speculative side-channels. As already mentioned, NDA [52] and STT [56] use the same principles, with NDA being more strict by not allowing potential secrets to propagate to any dependent instructions, achieving reduced instruction level parallelism, while STT applies a taint tracking mechanism to propagate secrets and delay only dependent transmitting instructions. DoM [39, 40], instead of tracking potential secrets and blocking transmitters, delays all loads that miss in the (L1) data cache, as hits do not produce timing effects. This eliminates all observable cache timing differences. Mechanisms such as InvisiSpec [53], Ghost Loads [38], MuonTrap [5], and GhostMinion [4] focus on hiding speculative execution by using speculative buffers that temporary store speculative information, and modifying the memory system to comply with this invisibility. CleanupSpec [37] focuses on restoring microarchitectural states after misspeculation is verified, effectively scrubbing potential secrets from the observable state. There have also been several attacks [2, 3, 9, 27] that target existing schemes, but ReCon does not either affect or enhance their effectiveness as they can be applied independently.

The above solutions introduce varying performance overheads and implementation complexities. To recover some lost performance, several optimizations have been proposed.

Speculative Data-Oblivious execution (SDO) [55] is an optimization to STT that uses prediction to make speculative execution independent of speculatively accessed values. In contrast to ReCon, SDO focuses on STT, and cannot be readily combined with other schemes as, for example, with NDA. NDA does not propagate the secret, and thus SDO would be unable to predict the cache level hit for dependent loads. SDO on STT provides a 44.4% reduction in overhead with the Spectre threat model and 36.3% with the futuristic threat model, when protecting against memory side-channels (load instructions). While our evaluation differs (e.g., we use SPEC2017 *speed* benchmarks instead of *rate*, and with an entirely different set of simpoints), we report a 45% reduction in overhead for a threat model that lies between the Futuristic and the Spectre threat model

(36.3% and 44.4% reduction in SDO, respectively). Not only that, but the two optimization mechanisms are orthogonal and can cooperate, as ReCon applies on untainted loads to untaint their dependent loads, while SDO applies on tainted loads to predict their cache level hit. Thus, they can both be applied at the same time, ReCon reducing the number of tainted load instructions and *SDO* recovering performance by predicting tainted loads.

Other optimizations: InvarSpec[58] detects load instructions that are guaranteed to commit regardless the outcome of speculation, lifting their protection while still speculative. InvarSpec operates together with secure speculation schemes that protect against all speculative data leakage (e.g., InvisiSpec [53]). While it can also be adapted for schemes like STT and NDA, the performance gains are unknown as those schemes already explore some memory-level parallelism by allowing independent loads to happen while speculative. This is a major performance bottleneck for DoM [39], for example, which delays all loads that miss in the first level cache and InvarSpec [58] enables the execution of some of those specific misses.

Clearing the Shadows [48] focuses on instruction re-ordering to eliminate speculation as early as possible. InvarSpec and Clearing the Shadows are optimizations that leverage compilers and hardware/software co-design, unlike our approach, which only affects the hardware implementation.

Pinned Loads [57] focuses on speculation and the overhead caused by memory re-ordering, proposing a mechanism to resolve memory violations as early as possible, to enable the execution of protected loads much earlier. Both Clearing the Shadows [48] and Pinned Loads [57] focus on eliminating speculation and thus boosting performance by assigning less work to the underneath mitigation. This is different from our work, where we actually try to optimize the existing schemes.

Doppelganger Loads [26] is an optimization that also leverages non-speculative information, but instead of directly connecting it to leakage (load pairs), it uses the addresses accessed by committed loads to train an address predictor and safely predict the address of subsequent speculative loads.

8 CONCLUSION

We propose ReCon, an efficient, low-complexity approach to leverage knowledge of non-speculative leakage for the purpose of relaxing defenses in secure speculation mechanisms, such as NDA and STT, that would otherwise protect data that have already leaked.

Based on the observation that an address accessed by a load and transmitted by a second dependent load, leaks the value at this address, ReCon focuses exclusively on detecting non-speculative, direct-dependent load pairs, shedding all the complexity of a general dynamic information flow tracking (DIFT) tracking mechanism proposed previously. Furthermore, ReCon leverages the existing cache coherence infrastructure (including the directory) to store, share, transmit, and keep coherent the non-speculative-leakage state of addresses. ReCon, depending on the underlying secure speculation mechanism, enables the execution of load instructions that would otherwise be delayed. For example, under STT, ReCon untaints the output register of the load instruction that accesses an address known to have leaked non-speculatively.

ReCon successfully reduces the overhead for NDA by 28.7%, and 31.5%, and the overhead for STT by 45.1%, and 39%, on average, for the SPEC2017, and SPEC2006. For the PARSEC benchmark suite, ReCon reduces the overhead incurred in the total execution time by 78.6%, and 46.7%, respectively for NDA and STT.

ACKNOWLEDGMENTS

This work was supported by the VINNOVA grant 2021-02422, by Microsoft Research through its EMEA PhD Scholarship Programme grant 2021-020, the Swedish Research Council (VR) grant 2018-05254, and the Swedish Foundation for Strategic Research (SSF) grant FUS21-0067.

The simulations were performed by resources in project NAISS 2023/22-3 provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) at UPPMAX, funded by the Swedish Research Council through grant agreement no. 2022-06725.

We thank the anonymous shepherd and reviewers for their valuable input.

REFERENCES

[1] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. 2009. GARNET: A detailed on-chip network model inside a full-system simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software.* 33–42. https://doi.org/10.1109/ISPASS.2009.4919636

[2] Pavlos Aimoniotis, Amund Bergland Kvalsvik, Magnus Själander, and Stefanos Kaxiras. 2022. Data-Out Instruction-In (DOIN!): Leveraging Inclusive Caches to Attack Speculative Delay Schemes. In *Proceedings of the IEEE International Symposium on Secure and Private Execution Environment Design.* 49–60. https://doi.org/10.1109/SEED55351.2022.00012

[3] Pavlos Aimoniotis, Christos Sakalis, Magnus Själander, and Stefanos Kaxiras. 2021. Reorder Buffer Contention: A Forward Speculative Interference Attack for Speculation Invariant Instructions. *IEEE Computer Architecture Letters* 20 (July 2021), 162–165. Issue 2. https://doi.org/10.1109/LCA.2021.3123408

[4] Sam Ainsworth. 2021. GhostMinion: A Strictness-Ordered Cache System for Spectre Mitigation. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture.* 592–606. https://doi.org/10.1145/3466752.3480074

[5] Sam Ainsworth and Timothy M. Jones. 2020. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. In *Proceedings of the International Symposium on Computer Architecture.* 132–144. https://doi.org/10.1109/ISCA45697.2020.00022

[6] A.R. Alameldeen and D.A. Wood. 2006. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro* 26 (July 2006), 8–17. Issue 4. https://doi.org/10.1109/MM.2006.73

[7] AMD 2023. AMD Zen 3 Microarchitecture. https://en.wikichip.org/wiki/amd/microarchitectures/zen_3

[8] AMD 2023. AMD Zen 4 Microarchitecture. https://en.wikichip.org/wiki/amd/microarchitectures/zen_4

[9] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. 2021. Speculative interference attacks: breaking invisible speculation schemes. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems.* 1046–1060. https://doi.org/10.1145/3445814.3446708

[10] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security.* 785–800. https://doi.org/10.1145/3319535.3363194

[11] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectural and Compilation Techniques.* 72–81. https://doi.org/10.1145/1454115.1454128

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39 (May 2011), 1–7. Issue 2. https://doi.org/10.1145/2024716.2024718

[13] Xiaoyue Chen, Pavlos Aimoniotis, and Stefanos Kaxiras. 2023. Clueless: A Tool Characterising Values Leaking as Addresses. *arXiv preprint arXiv:2301.10618* (Jan. 2023). https://doi.org/10.48550/arXiv.2301.10618 arXiv:arXiv:2301.10618

[14] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. 2021. Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 607–622. https://doi.org/10.1145/3466752.3480068

[15] G.Z. Chrysos and J.S. Emer. 1998. Memory dependence prediction using store sets. In *Proceedings of the International Symposium on Computer Architecture*. 142–153. https://doi.org/10.1109/ISCA.1998.694770

[16] Standard Performance Evaluation Corporation. 2006. SPEC CPU2006 Benchmark Suite. http://www.specbench.org/cpu2006/

[17] Standard Performance Evaluation Corporation. 2017. SPEC CPU2017 Benchmark Suite. http://www.specbench.org/cpu2017/

[18] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. 2023. SQUIP: Exploiting the Scheduler Queue Contention Side Channel. In *Proceedings of the IEEE Symposium on Security and Privacy*. 468–484.

[19] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. *arXiv preprint arXiv:2210.14324* (Oct. 2022). https://doi.org/10.48550/arXiv.2210.14324 arXiv:arXiv:2210.14324

[20] Intel 2022. *Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html

[21] Intel 2023. Intel Skylake Microarchitecture. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

[22] jannh@google.com. 2018. Issue 1528: speculative execution, variant 4: speculative store bypass - project-zero. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528

[23] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *arXiv preprint arXiv:1807.03757* (July 2018). arXiv:1807.03757 [cs] http://arxiv.org/abs/1807.03757

[24] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1–19. https://doi.org/10.1109/SP.2019.00002

[25] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *Proceedings of the USENIX Workshop on Offensive Technologies*. https://www.usenix.org/conference/woot18/presentation/koruyeh

[26] Amund Bergland Kvalsvik, Pavlos Aimoniotis, Stefanos Kaxiras, and Magnus Själander. 2023. Doppelganger Loads: A Safe, Complexity-Effective Optimization for Secure Speculation Schemes. In *Proceedings of the International Symposium on Computer Architecture*. 1–13. https://doi.org/10.1145/3579371.3589088

[27] Mengming Li, Chenlu Miao, Yilong Yang, and Kai Bu. 2022. unXpec: Breaking Undo-based Safe Speculation. In *Proceedings of the International Symposium High-Performance Computer Architecture*. 98–112. https://doi.org/10.1109/HPCA53966.2022.00016

[28] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. 2019. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *Proceedings of the International Symposium High-Performance Computer Architecture*. 264–276. https://doi.org/10.1109/HPCA.2019.00043

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the USENIX Security Symposium*.

[30] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2109–2122. https://doi.org/10.1145/3243734.3243761

[31] Andreas Ioannis Moshovos. 1998. *Memory Dependence Prediction*. Ph. D. Dissertation. University of Wisconsin.

[32] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: attacking ARM pointer authentication with speculative execution. In *Proceedings of the International Symposium on Computer Architecture*. 685–698. https://doi.org/10.1145/3470496.3527429

[33] Red Hat) 2022. Simultaneous Multithreading in Red Hat Enterprise Linux. https://access.redhat.com/solutions/rhel-smt

[34] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site Isolation: Process Separation for Web Sites within the Browser. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1661–1678. https://www.usenix.org/conference/usenixsecurity19/presentation/reis

[35] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. 2021. I See Dead μops: Leaking Secrets via Intel/AMD Micro-Op Caches. In *Proceedings of the International Symposium on Computer Architecture*. 14. https://doi.org/10.1109/ISCA52012.2021.00036

[36] Alberto Ros, Trevor E. Carlson, Mehdi Alipour, and Stefanos Kaxiras. 2017. Non-Speculative Load-Load Reordering in TSO. In *ACM SIGARCH Computer Architecture News*, Vol. 45. 187–200. https://doi.org/10.1145/3140659.3080220

[37] Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. CleanupSpec: An "Undo" Approach to Safe Speculation. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 73–86. https://doi.org/10.1145/3352460.3358314

[38] Christos Sakalis, Mehdi Alipour, Alberto Ros, Alexandra Jimborean, Stefanos Kaxiras, and Magnus Själander. 2019. Ghost loads: What is the cost of invisible speculation?. In *Proceedings of the ACM International Conference on Computing Frontiers*. 153–163. https://doi.org/10.1145/3310273.3321558

[39] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient Invisible Speculative Execution through Selective Delay and Value Prediction. In *Proceedings of the International Symposium on Computer Architecture*. 723–735. https://doi.org/10.1145/3307650.3322216

[40] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2020. Understanding Selective Delay as a Method for Efficient Secure Speculative Execution. *IEEE Trans. Comput.* 69 (Nov. 2020), 1584–1595. Issue 11. https://doi.org/10.1109/TC.2020.3014456

[41] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *Proceedings of the European Symposium on Research in Computer Security*. 279–299. https://doi.org/10.1007/978-3-030-29959-0_14

[42] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Michael Schwarz, and Daniel Gruss. 2022. Robust and Scalable Process Isolation Against Spectre in the Cloud. In *Proceedings of the European Symposium on Research in Computer Security*, Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng (Eds.). 167–186. https://doi.org/10.1007/978-3-031-17146-8_9

[43] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53 (July 2010), 89–97. Issue 7. https://doi.org/10.1145/1785414.1785443

[44] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. 45–57. https://doi.org/10.1145/605397.605403

[45] G Edward Suh, Jaewook Lee, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. *ACM SIGPLAN Notices* 39 (2004), 85–96. Issue 11.

[46] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. 2022. SecSMT: Securing SMT Processors against Contention-Based Covert Channels. In *Proceedings of the USENIX Security Symposium*. 3165–3182. https://www.usenix.org/conference/usenixsecurity22/presentation/taram

[47] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. 395–410. https://doi.org/10.1145/3297858.3304060

[48] Kim-Anh Tran, Christos Sakalis, Magnus Själander, Alberto Ros, Stefanos Kaxiras, and Alexandra Jimborean. 2020. Clearing the Shadows: Recovering Lost Performance for Invisible Speculative Execution through HW/SW Co-Design. In *Proceedings of the International Conference on Parallel Architectural and Compilation Techniques*. 241–254. https://doi.org/10.1145/3410463.3414640

[49] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2017. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. *ACM SIGARCH Computer Architecture News* 45 (March 2017), 119–133. Issue 1. https://doi.org/10.1145/3093337.3037719

[50] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1995. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the International Symposium on Computer Architecture*. 392–403. https://doi.org/10.1145/223982.224449

[51] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. 2022. Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1491–1505. https://doi.org/10.1109/SP46214.2022.9833570

[52] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 572–586. https://doi.org/10.1145/3352460.3358306

[53] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 428–441. https://doi.org/10.1109/MICRO.2018.00042

[54] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. 2018. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. *Cryptology ePrint Archive* (2018). https://eprint.iacr.org/2018/808

[55] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. 2020. Speculative Data-Oblivious Execution: Mobilizing Safe Prediction For Safe and Efficient Speculative Execution. In *Proceedings of the International Symposium on Computer Architecture*. 707–720. https://doi.org/10.1109/ISCA45697.2020.00064

[56] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 954–968. https://doi.org/10.1145/3352460.3358274

[57] Zirui Neil Zhao, Houxiang Ji, Adam Morrison, Darko Marinov, and Josep Torrellas. 2022. Pinned loads: taming speculative loads in secure processors. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. 314–328. https://doi.org/10.1145/3503222.3507724

[58] Zirui Neil Zhao, Houxiang Ji, Mengjia Yan, Jiyong Yu, Christopher W. Fletcher, Adam Morrison, Darko Marinov, and Josep Torrellas. 2020. Speculation Invariance (InvarSpec): Faster Safe Execution Through Program Analysis. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 1138–1152. https://doi.org/10.1109/MICRO50266.2020.00094