Architectural Contracts for Safe Speculation

Franz A. Fuchs*, Jonathan Woodruff*, Peter Rugg*, Marno van der Mass*, Alexandre Joannou*,
Alexander Richardson*, Jessica Clarke*, Nathaniel Wesley Filardo§, Brooks Davis†,
John Baldwin‡, Peter G. Neumann†, Simon W. Moore*, Robert N. M. Watson*

*Department of Computer Science and Technology, University of Cambridge, Cambridge, UK
†SRI International, Menlo Park, CA, USA ‡Ararat River Consulting, Walnut Creek, CA, USA
§Microsoft Research Ltd, Cambridge, UK
{franz.fuchs, jonathan.woodruff, peter.rugg, marno.van-der-maas, alexandre.joannou}@cl.cam.ac.uk
{alexander.richardson, jessica.clarke, simon.moore, robert.watson}@cl.cam.ac.uk
{neumann,brooks}@csl.sri.com john@araratriver.com nfilardo@microsoft.com

Abstract—**We propose architectural contracts that specify the allowable limits of speculative execution to enable both software safety guarantees and hardware verification. Transient-execution attacks have presented a major threat in recent years, driving deployment of software mitigations and research into hardware solutions. Recent work on hardware/software contracts for secure speculation recognizes the need for cooperation between hardware guarantees and software analysis, and demonstrates that speculative execution models can enable formal analysis of programs with respect to transient-execution vulnerabilities. Therefore, we have extended these limited models into comprehensive architecture-level contracts that can be verified at a microarchitecture level. We define a set of speculation contracts for translation (TSC) and branching (BSC), and for memory ordering (MOSC). We also develop a set of directed-random test routines that reproduce all known contract violations in a prototype out-of-order processor, most of which represent known transient-execution vulnerabilities. We also extend the RiscyOO processor to enforce each contract and evaluate performance, demonstrating the practicality of the chosen contracts with an overhead between -1.2% and +1.8% for this prototype. These general-purpose contracts set the stage for specification of speculative execution for complete instruction-set architectures, and particularly for new security-focused ISA extensions.**

Index Terms—**transient-execution attacks, instruction-set architectures, computer security, hardware-software contracts, testing, microarchitecture**

I. Introduction

Transient-execution attacks [1]–[5] have greatly impacted the computer security landscape. Transient-execution attacks aim to encode secrets in microarchitectural state changes, which are then made visible via side channels (e.g., cache timing attacks [6]) to leak information between protection domains. While some transient-execution vulnerabilities are fixable in hardware [1], [7], others require long-term software mitigations. In both cases, instruction-set architectures (ISAs) suffer from a specification vacuum with respect to behavior in speculation. On the one hand, hardware designers are unclear to what degree they are responsible for transient-execution side-channel attacks. On the other hand, software mitigations are being developed around ad hoc mental models of speculative execution [8]. These mental models are unlikely to be complete for any one microarchitecture, and certainly not for all current and future implementations. This leads to mitigations that fail in unexpected ways [9].

Speculation contracts have been proposed to define program safety with respect to transient-execution side-channel attacks [10]. This work has demonstrated that a well-defined (if limited) execution model can support reasoning about the visibility of state to observers of transient execution. We propose a complete speculation model for execution, in the form of a set of contracts for the instruction-set architecture, forcing implementations to confine speculation within limits that allow software analysis. These ISA-level contracts do not categorically prevent side channels, but enable sound reasoning about software with respect to safety against classes of transient-execution attacks. We propose:

1) The Translation Speculation Contract (TSC) to constrain which virtual addresses may be used to speculatively access memory.
2) The Branching Speculation Contract (BSC) to constrain which control-flow paths may execute in speculation.
3) The Memory Ordering Speculation Contract (MOSC) to constrain which values may be speculatively loaded from an address.

These architectural speculation contracts must allow for practical hardware verification in order to be meaningful to microarchitectural design. To prove that our contracts are testable, we have developed an automated testing framework to discover speculation contract violations in processors.

In this work, we answer the following research questions:

1) *Is it possible to define useful architectural speculation contracts?* In Section IV, we define exemplar architectural speculation contracts to support reasoning about translation, control flow, and memory ordering.
2) *How can we test architectural speculation contracts?* We show in Section VIII that testing implementations against architectural speculation contracts is practical. We develop a portable testing strategy with an automated

This project was supported by the NCSC under the UK RISE Initiative. This work was supported in part by the Engineering and Physical Sciences Research Council EP/S030867/1. Distribution Statement A: Approved for public release; distribution is unlimited. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract HR0011-18-C-0016 ("ECATS"). The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

tool for directed-random testing for conformance to the above speculation contracts. Our tests flag multiple violations that uncover challenges in implementing even straightforward contracts.
3) *Can we enforce speculation contracts?* In Section IX, we demonstrate that architectural speculation contracts can be enforced in modern microarchitectures. We implement contract enforcement mechanisms in a superscalar, out-of-order processor and identify common mechanisms that could be useful in other microarchitectures.
4) *Can speculation contracts be enforced with reasonable performance?* In Section X, we demonstrate that architectural speculation contracts can be implemented with near-zero overhead in SPEC2006 [11].

II. The Architectural Threat Model

Security design requires a clearly defined threat model. We can consider security design in computer systems at three levels: software, instruction-set architecture, and processor microarchitecture. While software programs often have clear use cases with defined threat models, instruction-set architectures are necessarily general. Consequently, the most widespread ISA-level threat models are simple: preserve privilege rings and isolate address spaces. At the bottom of the stack, processor microarchitectures have required almost no security design beyond implementing the ISA specification; security vulnerabilities were simply architecturally-visible bugs.

Transient-execution attacks have greatly disturbed security design in computer systems by assigning blame to processor microarchitects for non-architectural timing behavior, which was indeed used to bypass architectural privilege rings and address space isolation. To maintain an orderly chain of trust, we suggest that instruction-set architectures now consider threats from attackers who can wield transient execution, and take up the burden of defining the limits of safe speculation. These architectural limits should be proven capable of preventing transient side-channel threats with reasonably structured software. This full definition of the limits of speculative execution would allow microarchitects to return to a threat model that can be reduced to an ISA test suite.

For the instruction-set architecture itself, we now require a more nuanced threat model. While architectures previously required that secret values must not be accessible through memory or through an architectural register, in this work we will assume that any value that can enter *the core* (defined in Section IV-A) in speculation is accessible through a side-channel. This allows us to avoid consideration of the multitude of side-channel mechanisms, which are comparatively harder to specify than speculative memory accesses and execution paths that can be expressed in terms of the architecture.

III. Background and Motivation

Recent history with transient-execution side channels proves the need for clear specification of the limits of speculative execution to support both software reasoning and hardware development.

A. Hardware Side-Channel Defense

Early work on mitigating transient-execution attacks in hardware focused on closing cache timing side-channels [12]. This was followed by Speculative Taint Tracking [13] and Speculative Privacy Tracking [14], which tracked and limited speculation to avoid exposing transient data to side-channels. These protections incur substantial overhead, begging the question of what minimal properties software requires to be provably safe. In a precursor to speculation contracts, SpecTerminator [15] improves on previous work by defining allowed speculation in one case. This shows the need for general ISA-level contracts that enable software to reason about its security guarantees.

B. Speculation in Current Architectures

While speculative behavior is generally left unspecified in current instruction-set architectures (ISAs), common architectures do provide rudimentary barriers. The x86-64 ISA specifies that the *lfence* instruction prevents speculative out-of-order execution [16], and Arm defines speculation barriers, e.g., the `SSBB` instruction [17]. Furthermore, Arm proposes architectural extensions to allow software to entirely disable categories of speculation or to keep speculation in one context exclusively [18]. These speculation barriers and mechanisms have provided a crucial handle in ISA specifications that allow software mitigations for transient-execution attacks. However, the lack of general guarantees on speculative execution poses great risks, and will inevitably result in overly conservative hardware and software. Barriers in hardware must prevent all speculation, and these barriers will be used liberally by software to prevent esoteric potentialities, as the ISA provides no other guarantees on behavior in speculation. This conundrum is similar to memory-consistency models, where lack of specification results in overuse of memory barriers [19]. Memory model specification may at first appear unnecessary in an ISA, but has been proven to be necessary for portable concurrent algorithms. In the same way, portable, high-performance mitigations of transient-execution attacks require some guaranteed limits on speculation.

C. Contracts for Secure Speculation

Hardware-Software Contracts for Secure Speculation [10] have proposed a framework comprised of execution models and side-channel observers. This framework facilitates program analysis to identify transient-execution side channels in the combination of processor speculation and program structure [20]. We complement this work by extending from simplistic speculation models to full architectural speculation contracts to provide a foundation for reasoning about program safety using their proposed style of software analysis and vulnerability discovery.

IV. Speculation Contracts

Our architectural speculation contracts target the overlap between high-performance processor design and reasonable semantics for specification. As introduced in Section II, our contracts focus on speculative-execution possibilities rather than side-channel prevention.

A. Common Vocabulary

Our contracts are presented in prose for accessibility. However, we must clearly define our use of the following terms:
A *compartment* is a unit of software decomposition expressed architecturally to be enforced by hardware. Examples might be privilege level (expressed in rings) or address spaces (expressed in process IDs or page table roots). Future architectures might allow more fine-grained compartments. Many system might do compartmentalization on a process-level, where each process becomes an own compartment as well as making the kernel its own compartment. When doing a compartment switch, e.g., a system call changing from a user space compartment to the kernel compartment, the microarchitecture needs to be made aware of the compartment change. Future research needs to investigate ways of performing microarchitectural compartment changes.
We define *the core* as reaching from the execution pipeline to the load/store queue, but not to the L1 caches. This model assumes unused bytes of cache lines cannot leak, and that the cache itself, including any prefetching, does not perform data-dependent operations sufficient to create side-channels. We also ignore asynchronous agents like the page-table walker and DMA engines.
An instruction is *fetched* if it is in the front-end of the processor. The front-end is characteristically in-order and flushed at lower cost than the out-of-order back end. Instructions that are merely fetched need not modify predictor state.
An instruction is *executed* when data-dependent action is taken using its operands, potentially producing a new data value in *the core*. In this stage, the instruction might modify predictor state and access data caches.
A *branch* is an instruction that conditionally affects control flow. This includes conditional branches to static targets and jumps to dynamic addresses, but not exceptions or interrupts.
An event is *non-speculative* if all previous branches have been executed.
An event is *transient* if it occurred speculatively but did not commit. A non-speculative event may still be transient.

B. Architectural Independence of Contracts

The contracts presented in the following subsections are architecture-independent and can be mapped to any ISA. Our contracts argue about general architectural principles not specific to any particular ISA. In the following subsections, we perform the step of mapping the contracts to RISC-V, and give examples of which scenarios are allowed and disallowed. It is future research to map our contracts onto other architectures.

C. Translation Speculation Contract

The Translation Speculation Contract (TSC) guarantees that architectural memory translation and page permissions are enforced continuously in the microarchitecture, including all speculative execution. This ensures that no unmapped memory access is issued from the core. Stated precisely:

Translation Speculation Contract (TSC)

All instruction and data-memory operations issued in speculation must be mapped in the page table, and allowed by current page permissions.

The Meltdown vulnerability [1] violates TSC, and was considered an egregious violation of programmers' expectations. Meltdown has not been considered endemic of high-performance microarchitectures; many high-performance implementations do not share this vulnerability and newer implementations avoid it without performance penalty [7]. Therefore, we conclude that the Meltdown vulnerability could be attributed purely to a lack of this architectural speculation contract.

D. Branching Speculation Contract

Control flow is the biggest challenge for a complete definition of speculative behavior. Instruction Fetch is the earliest stage of the pipeline and must operate with minimal knowledge of the instructions that are ostensibly directing it. Branch prediction is therefore traditionally the most speculative aspect of microarchitecture.

In order to define useful constraints for speculative control flow, we strategically limit which instructions will be *executed* rather than merely *fetched*. Side channels from speculative execution are useful when they can exfiltrate an illegal value; the production of an illegal value generally happens in the Execute stage of the pipeline, e.g., when a load is actually performed. This allows the front-end of the pipeline to speculate freely until decoding the instruction stream. Allowing wild instruction fetch assumes that it is not possible to leak the contents of memory through the front-end of the pipeline. This should at least require constant-time decoding. After Instruction Decode, the microarchitecture can intelligently enforce architectural speculation contracts with knowledge of the instructions that would be executed.

The Branching Speculation Contract (see definition) is designed to allow reasoning about the possibilities of speculative execution within a compartment, and also to provide guarantees between compartments.

BSC item 1 only allows execution to follow the correct target of a direct jump. In this case the target is known in Decode and mispredictions can be flushed before execution. This allows basic blocks to span direct jumps. For example:

```
labelA:
    jal x0, labelB
    ...
labelB:
    ...
```

The direct jump can proceed only to `labelB` – no other speculatively executed path is allowed.

Item 2 precludes the processor from conducting wild target speculation for direct branches. For example:

Branching Speculation Contract (BSC)

1) Direct jumps (unconditional with immediate target) will be followed in speculative execution only by the jump target.
2) Direct branches (conditional with immediate target) will be followed in speculative execution only by one of the two possible targets.
3) Instructions that cause exceptions will be followed in speculative execution only by the non-faulting path.
4) Indirect branches (register target) will be followed in speculative execution only by previous architectural indirect-branch targets from the current compartment.
5) Returns will be followed in speculative execution only by the instruction after a previous call from the current compartment.

```
    blt x6, x7, labelA
    ...
labelA:
    ...
```

After the conditional branch, speculative execution can proceed only to the next instruction, or to the instruction at `labelA`. This constraint is also foundational for current Spectre mitigations, which assume that control flow at a direct branch might take the wrong path due to a misprediction in the Pattern History Table (PHT), but will not select an arbitrary target from the Branch Target Buffer (BTB). Direct branches are nevertheless likely to use a simple BTB for all early branch predictions, but microarchitectures should not proceed to execution with a conditional branch prediction that is not one of the two allowed targets.

Item 3 guarantees that exception paths will never be speculatively initiated by the processor, and that the straightforward non-faulting path will always be taken in speculative execution. While it could be tempting to speculate on exceptions such that repeated exceptions will train the processor to speculatively enter the exception handler, we recommend this simpler constraint. Exceptions are not a common case, and mixing instructions of different privilege levels in speculative execution increases the reasoning burden for software.

With item 4, only previously committed targets from this *compartment* may be predicted, allowing reasoning about the complete set of potential speculative targets of an indirect branch. Predicting targets learned from transient execution would make static analysis impossible. In fact, we find in Section X that excluding transient targets in the BTB can improve performance. Item 4 also prevents indirect branches from defaulting to straight-line prediction, which has been a source of transient-execution attacks [21].

Item 5 describes standard call stack prediction using the return stack buffer (RSB), but allows training only from non-speculative function calls. This is expected behavior, and forms the basis of the broadly deployed Retpoline Spectre-BTB mitigation recommended by Intel [22]. Nevertheless, this expectation is not specified architecturally; both Intel and AMD processors use BTB predictions for the call stack in certain cases, allowing Retpoline mitigations to be bypassed [9].

Constraining speculation training to non-speculative jumps from the same compartment, as specified in items 4 and 5, enables full analysis of possible speculative paths in a compartment, and also guarantees non-interference between compartments.

V. Data-value Speculation

Data-value speculation is used in modern processors in various forms. Memory is the most important target for data-value speculation, and speculatively reordering loads and stores is a basic feature of modern out-of-order cores. For this initial work, we present the Memory Ordering Speculation Contract (MOSC) as a complete data-value speculation contract; no other form of data-value speculation is allowed.

Memory Ordering Speculation Contract (MOSC)

Speculatively executed loads will return a value that has been held in that address while in this compartment. Speculatively stored values may be loaded from this compartment; only committed (non-transient) values may be loaded from other compartments.

These semantics of MOSC are a compromise between architectural semantics and microarchitectural freedom. MOSC allows the most natural form of speculative memory reordering without compromising compartmentalization as long as software is careful about memory initialization.

VI. Security Evaluation

Architectural speculation contracts embrace an ISA threat model that includes transient-execution attacks in the effort to improve architecture-wide security guarantees. In this section, we evaluate the security guarantees of our proposed contracts in the face of transient-execution attacks. We follow the classification of attacks from Canella et al. [23].

A. Spectre Defense

Spectre-like attacks are classified based on the prediction mechanism they exploit: Spectre-PHT, Spectre-BTB, Spectre-RSB, and Spectre-STL (Store-to-Load forwarding).

Whereas BSC limits direct branch-direction prediction to the two architectural possibilities via constraint 2, it does not natively prevent Spectre-PHT attacks that use a transient wrong path to bypass security checks. However BSC does guarantee that common software mitigations, such as pointer masking, will be effective.

BSC limits the speculative targets of indirect jumps, calls, and returns to committed targets from the same compartment

	Intra-Compartment	Inter-Compartment
Spectre-PHT	✗	✗
Spectre-BTB	✓(limited)	✓
Spectre-RSB	✓(limited)	✓
Spectre-STL	✗	✓

TABLE I: Spectre security evaluation for intra-compartment and inter-compartment attacks.

as a defense against Spectre-BTB and Spectre-RSB. Attackers can only train the Branch Target Buffer (BTB) and Return Stack Buffer (RSB) with architectural targets, so cannot redirect control flow to arbitrary targets. Furthermore, BSC does not allow an attacker to train their own targets into another compartment. However, BSC constraints 4 and 5 do not forbid wrong target speculation as long as a jump to that target has been committed previously in this compartment. While an attacker cannot inject their own targets, an attacker might still mistrain another compartment, e.g., through BTB conflicts, to misspeculate to local targets that may reveal secrets. Nevertheless, BSC does enable a compartment to enumerate all potential speculative targets of an indirect branch, by analyzing only its own code.

Spectre-STL attacks mistrain the memory disambiguation predictor, with the goal of transiently loading secret data to be exfiltrated via a side channel. MOSC prevents inter-compartment Spectre-STL attacks if secrets are cleared before memory is shared between domains.

As summarized in Table I, our contracts limit the inter-compartment attack surface for Spectre attacks.

B. Meltdown Defense

Meltdown-US illegally reads secrets from kernel pages that are still mapped in the page table – despite permissions prohibiting the access. TSC is an architectural formalization of hardware Meltdown-US protections and fully mitigates Meltdown-US attacks. TSC also prevents Foreshadow [24], a variant of Meltdown targeting Intel SGX encrypted pages.

Many ISA-specific Meltdown-like attacks have also been developed [23], e.g., Meltdown-NM [25] for x86 floating point. Most of those attacks are specific to one ISA and particular specific subsets of it. Our contracts are instruction-set architecture independent so do not consider such ISA-specific attacks. Including ISA-dependent contracts in our set would pollute the architectural interface.

C. Microarch. Data Sampling Defense

Microarchitectural data sampling (MDS) attacks harvest secrets by exfiltrating undefined data returned in the microarchitecture during failed speculation [26], [27]. Both TSC and MOSC prevent MDS attacks. TSC prevents MDS attacks against data forbidden by memory translation or permissions. MOSC also prevents MDS attacks by excluding all data-value speculation (including undefined data) besides memory reordering, which must return data belonging to the same address and compartment.

VII. RiscyOO Microarchitecture

To verify that these contracts could be implemented and tested, we used the RiscyOO processor. RiscyOO is a configurable out-of-order superscalar core in the Bluespec SystemVerilog language [28]. We have built on the Toooba fork of RiscyOO, which adds compressed instructions and debug support. We tested RiscyOO in the *SMALL* configuration, which can fetch and commit two instructions per cycle and has an out-of-order window of 64 instructions. This configuration has two integer pipelines, one memory pipeline, and one floating-point pipeline, each fed from a 16-entry issue queue. The memory queues track 38 outstanding memory transactions; the L1 caches are 32 KiB and 8-way associative, and the L2 cache is 1 MiB and 16-way associative.

Transient-Vulnerability Profile: All basic Spectre attacks are possible on RiscyOO [29], [30], and the reorder window proved more than sufficient for a side-channel gadget to execute while waiting for slow-to-resolve misspeculation. Despite our SMALL configuration, we found that an indirect branch that is waiting on its target to return from memory can allow more than 28 predicted instructions to execute before flushing on misprediction. This enables exploits that inject cache side-channel gadgets using the BTB and RSB, as well as simpler attacks using the PHT.

RiscyOO is also vulnerable to BTB aliasing attacks, particularly when using hashed BTB tags.

The PHT of RiscyOO is also vulnerable to mistraining, but only to one of the legitimate targets of the branch [2], [29], [30].

RiscyOO's sophisticated memory subsystem is also vulnerable to Spectre Store-To-Load (STL).

The range of Meltdown attacks could not be reproduced in RiscyOO, because page permissions are checked before issuing to memory (Meltdown-US [1]) – as is common [17], and because system register permissions are checked in the Rename stage before instructions are issued (Meltdown-GP).

VIII. Testing Speculative Contracts

As with all components of an ISA, architectural speculation contracts should be verifiable.

The IntroSpectre verification effort shows the difficulty faced by the current state of the art [31]; IntroSpectre uses debug tracing in simulation and directed-random test generation to discover leakage of state between privilege levels. The IntroSpectre approach is inconclusive (because the expected behavior is undefined) and also non-portable (because it relies on non-architectural debug trace of one implementation).

We intend to provide a clear set of contracts against which to test, and a portable testing strategy. For a portable testing approach, we relied on the RISC-V Formal Interface (RVFI), a standard architectural tracing format used by the TestRIG framework for RISC-V processors [32]. Furthermore, we used the Hardware Performance Monitoring (HPM) extension to detect microarchitectural events; HPM counters provide a portable bridge between architectural state and the speculative behavior guaranteed by our contracts. Whereas arranging scenarios where counters reliably indicate violations was sometimes complex, this approach is portable to a family of implementations that share the same architectural contracts.

A. Translation Speculation Contract Testing

The Translation Speculation Contract requires that transient execution respects page-table mappings and permissions. Our TSC generator populates a page table with one code and one data page, and arranges for the data-cache miss counter to register any accesses outside of these pages. Random memory instructions then attempt to access disallowed pages. As expected from the results of previous research [29], [30], our generator did not detect a TSC violation.

B. Branching Speculation Contract Testing

BSC consists of five constraints that we test via two generators: *branches and jumps* (constraints 1, 2, 4 and 5), and *exceptions* (constraint 3).

Branches and Jumps Violations: Our branch-and-jump test generator seeks to detect violations of BSC constraints 1, 2, 4, and 5. This ensures that direct jumps speculate only to the intended target; direct branches speculate only to one of the two possible targets; indirect jumps speculate only to a previous jump target; and returns speculate only to previous calls. Testing these properties in RiscyOO required implementing specialized counters using a small amount of state.

We detect direct-jump and direct-branch violations using custom assertions with counters in the Execute stage; these assertions register an event if the predicted nextPC is an implausible target.

For indirect jumps, we implemented a searchable vector of previously committed indirect jump targets and count *wild* predictions that are not found in this set. For returns, we similarly record previous call sites and count predicted return targets that are not in this set.

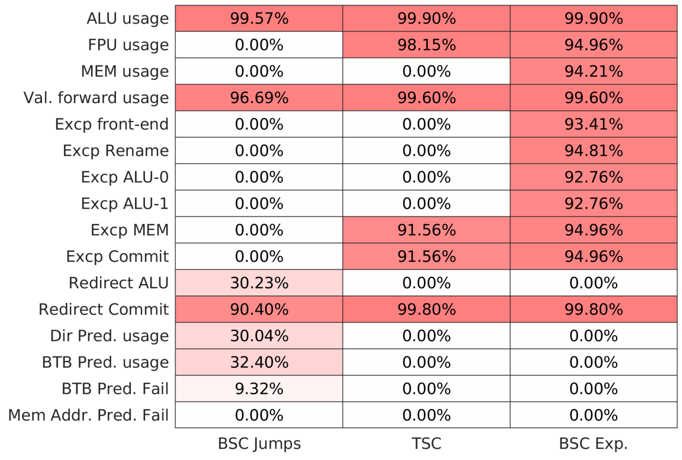
To prevent overflow of our structures, our generators ensure a limited number of jump targets by creating cycles of instructions with jumps, branches, or targets at fixed offsets.

Any branch that does not conform to our expectations for BSC is considered a *wild jump*, and is counted in our wild-jump counter. Our generator produced the following branch target buffer (BTB) speculative training violation:

```
csrci mcountinhibit (0x320), 8
jalr x1, x10, 21
srliw x28, x29, 1
```

The CSR Clear Immediate (`csrci`) instruction is a system instruction that causes a pipeline flush in RiscyOO. `jalr` speculatively executes after the `csrci`, but is flushed from the pipeline when `csrci` commits. Since the first execution of `jalr` does not commit, it does not produce a bonafide previous architectural target. Nevertheless, the transient `jalr` trained the BTB such that the `srliw` target executes speculatively after the flush, violating BSC constraint 4.

A return that violates BSC is considered a *wild return*, and is counted in our wild-return counter. Our generator produced



	BSC Jumps	TSC	BSC Exp.
ALU usage	99.57%	99.90%	99.90%
FPU usage	0.00%	98.15%	94.96%
MEM usage	0.00%	0.00%	94.21%
Val. forward usage	96.69%	99.60%	99.60%
Excp front-end	0.00%	0.00%	93.41%
Excp Rename	0.00%	0.00%	94.81%
Excp ALU-0	0.00%	0.00%	92.76%
Excp ALU-1	0.00%	0.00%	92.76%
Excp MEM	0.00%	91.56%	94.96%
Excp Commit	0.00%	91.56%	94.96%
Redirect ALU	30.23%	0.00%	0.00%
Redirect Commit	90.40%	99.80%	99.80%
Dir Pred. usage	30.04%	0.00%	0.00%
BTB Pred. usage	32.40%	0.00%	0.00%
BTB Pred. Fail	9.32%	0.00%	0.00%
Mem Addr. Pred. Fail	0.00%	0.00%	0.00%

Fig. 1: Microarchitectural event coverage by generator.

one class of counterexamples, which is caused by return stack buffer (RSB) training in speculation.

Exception Violations: BSC requires exceptions to speculate the non-faulting path. We added a counter for faulting instructions that predict a PC other than next instruction, but did not record any events; BSC constraint 3 is held by RiscyOO.

C. Memory Ordering Speculation Contract Testing

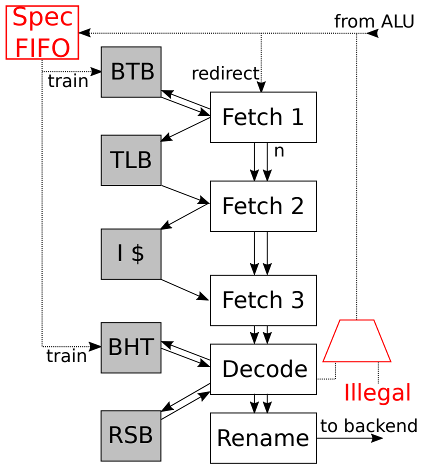
MOSC requires that loads only return values that have been previously held in that address. We introduced a small amount of state to track past stores, and count a wild load if a load returns an unseen value. We initially fill memory with zeros so that we can allow either zero or a recorded previous store value. Our generator creates random sequences of loads and stores intermixed with instructions from other classes. This generator did not find a counterexample, indicating that RiscyOO upholds MOSC.

D. Evaluation of Test Generators

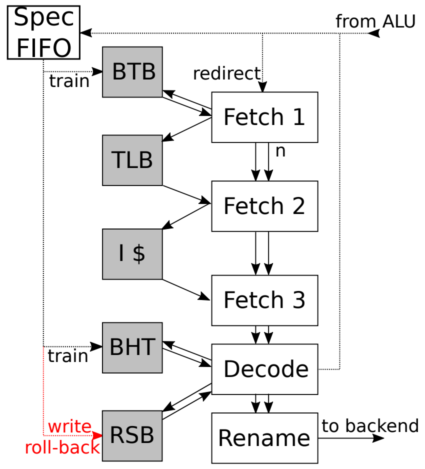
We traced microarchitectural events triggered by each test (Figure 1). For example, we found that 91.56% of the TSC tests triggered a memory exception due to insuffcent privileges, and 9.32% of all BSC Jump tests generated a BTB misprediction. At least 94.96% of BSC exception tests raised an exception, and these are categorized into front-end, Rename, ALU, or memory exceptions.

IX. Speculative Contract Enforcement

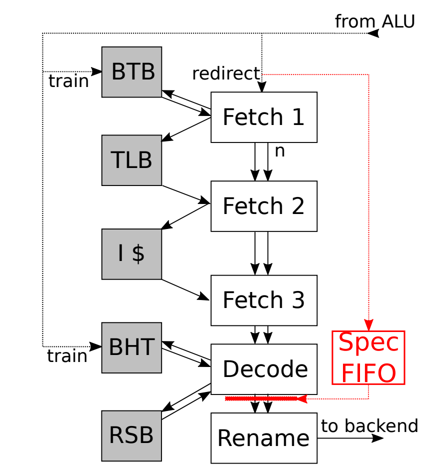
RiscyOO natively enforced TSC at full performance(see Section VIII-A). Enforcing the Branching Speculation Contract (BSC) on RiscyOO required delaying predictor training until the relevant branch was *non-speculative*, i.e., all previous branches have executed. To prevent mistraining predictors during an exception, we would use coarser measures (such as flushing) that are a secondary concern for common-case performance. In comparison with more direct side-channel prevention techniques such as those facilitated by Speculative Taint Tracking [13], this limitation is quite subtle and can



(a) The SpecFIFO component allows training the BTB and BHT with only non-speculative updates. The Decode stage can predict a guaranteed invalid address if no previous target is available. Combined, these help to enforce BSC constraint 4.



(b) The RSB can be rolled back on redirect, preventing desynchronization caused by speculative execution. In addition, pushing/popping can be deferred until the call/return is non-speculative. These target BSC constraint 5.



(c) A zero-data SpecFIFO allows instructions to be held at Decode until the instruction that caused the redirection is non-speculative. This is required to enforce BSC constraints 4 and 5.

Fig. 2: Changes made to the RiscyOO front-end to enforce BSC constraints.

yield a performance improvement. Waiting for state to be non-speculative is both high-performance and convenient for this implementation.

Our BSC enforcement in RiscyOO was facilitated by the *SpecFIFO* component from the RiscyOO code base, which holds a queue of speculative state along with tags that indicate which in-flight branches this state depends on. A branch resolution broadcast network ensures that metadata in SpecFIFOs are updated such that state belonging to failed speculation immediately disappears from all queues. A tag value of 0 indicates when this state is no longer speculative.

BSC constraints 1, 2, and 3 were already enforced in RiscyOO. A misprediction of a direct jump is detected in Decode when its target is fully known. Direct branches use a dedicated direction predictor in Decode to predict one of only two valid targets. If Decode detects an implausible prediction, it flushes the fetch pipeline and redirects to a legal target.

BSC constraint 4 mandates that all speculative indirect jump targets must have been previously targeted by committed jumps. This requires the Branch Target Buffer (BTB) to be trained only from *non-speculative* state. To enforce this, we converted the branch training queue into a SpecFIFO. This is a component in the RiscyOO base, which holds a queue of speculative state along with tags that indicate which in-flight branches this state depends on. This branch training SpecFIFO, shown in Figure 2a, is dequeued only when the head is *non-speculative*. This mechanism guards not only the BTB, but also the direction data in the Pattern History Table (PHT). Protecting direction prediction is not essential for BSC, and could have both positive and negative effects on branch accuracy due to delaying training and eliminating training from transient state. Further research needs to show the precise effect of BSC on branch predictor accuracy.

In addition, BSC constraint 4 forbids indirect jumps to default to straight-line prediction. Unfortunately, RiscyOO defaults to straight-line prediction for indirect jumps when no valid prediction was found in the BTB, which was found to be an exploitable vulnerability in commercial processor implementations [21]. To resolve this, we respond to a missing prediction by pausing fetch until we receive an actual redirection from Execute, as depicted in Figure 2a.

BSC constraint 5 requires that all speculative returns arrive at a previous non-speculative jump target. RiscyOO both pushed and popped the Return Stack Buffer (RSB) in Decode from the speculative instruction stream. That is, when a *Jump and Link* was encountered in Decode, PC+4 was pushed to this stack, and when a return was encountered, the top of the stack was popped and predicted as the next PC. This violates BSC constraint 5, as jumps that have pushed to the RSB might be flushed due to misprediction, introducing non-architectural targets to transient execution. In addition, performance was lost, as flushed transient instructions could cause the RSB stack to become desynchronized with the actual instruction stream. We resolved this with two mechanisms, illustrated in Figure 2b. First, we roll-back the RSB stack pointer on redirection; this reduced RSB mispredictions by roughly 90% in CoreMark. Second, we can delay RSB writes in the training SpecFIFO so that only *non-speculative* targets are written to the RSB stack. This is the most expensive aspect of our BSC enforcement, and is therefore listed separately in Figure 3.

Separate from training data, redirections themselves also needed protection to enforce BSC constraints 4 and 5. A redirected instruction stream in RiscyOO was able to reach execution while the instruction that caused the redirection was

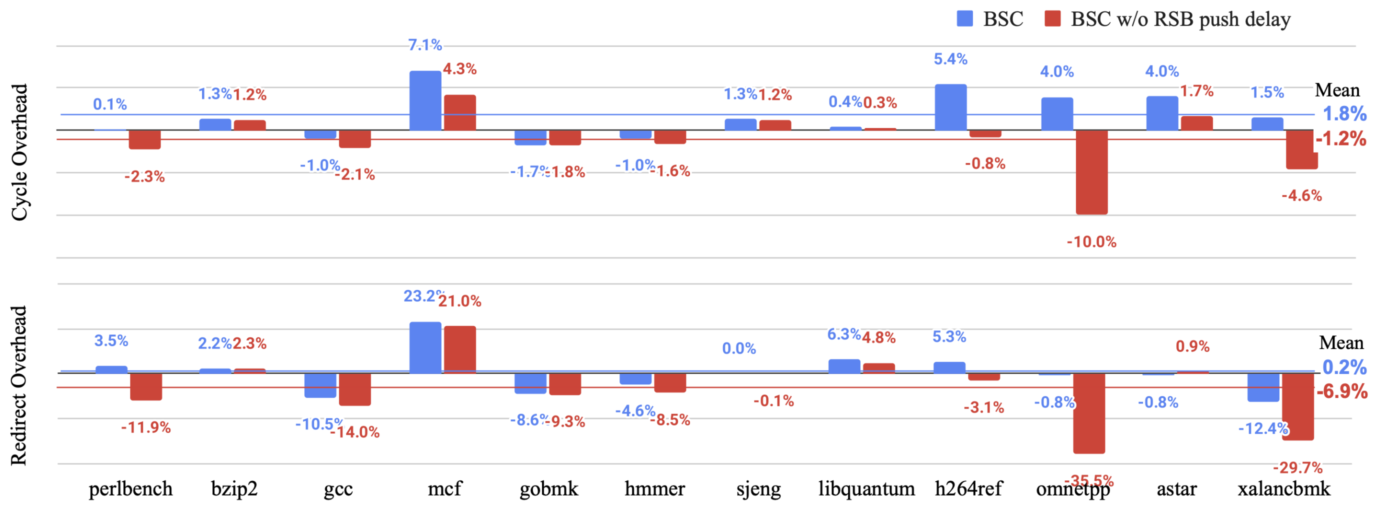


Fig. 3: Performance cost of enforcing the Branching Speculation Contract (BSC).

still speculative. This allows illegal transient targets to execute despite not populating either the BTB or the RSB.

The most straightforward solution would be to store the redirection itself in a SpecFIFO until the mispredicted branch that triggered the redirection had become non-speculative. However, this approach would have increased the misprediction penalty in many cases. Because BSC guarantees only the semantics of speculative *execution*, as opposed to fetch, Instruction Fetch is allowed to proceed with a speculative redirection, but Decode is blocked until the instruction that caused the redirection has become *non-speculative* (see Figure 2c). This is enforced by a zero-data SpecFIFO that holds the speculative state of the last redirection. This mechanism appears to block only very rarely, as performance is hardly affected by this change.

We tested these BSC enforcement mechanisms using the generators described in Section VIII, which helped identify stray violations and also helped ensure the test routines themselves were flagging appropriate violations. This exercise demonstrated that architectural speculation contracts of this style can be productively tested during development.

X. Performance Evaluation

Figure 3 gives the performance overhead of enforcing BSC on SPECInt2006 benchmarks [11]. We have measured both with and without the Return Stack Buffer (RSB) push delay, as this incurs the greatest overhead.

Without the RSB push delay, BSC protections reduce execution time by 1.2%. The performance improvement is due to reduced redirections from more accurate predictions, as seen in the lower graph. These more accurate predictions are due to BSC protections preventing prediction state from being polluted by transient instructions. While transient training sometimes provides positive hints for future behavior, in these cases they more often hurt performance, and there is a mean 6.9% reduction in redirections without the RSB push delay.

With RSB push delay enabled, there is a mean 1.8% cycle overhead. This feature increases redirections due to mispredicted short function calls, as mentioned in Section IX.

We were able to enforce the primary constraints of BSC in RiscyOO with near-zero overhead. Given the success of these rudimentary enforcements, we expect this result to generalize to more mature efforts in other microarchitectures.

Conclusion

This paper proposes architectural speculation contracts for instruction sets as a foundation for reasoning about the transient-execution vulnerability of programs. The contracts we have developed resolve the most egregious transient-execution vulnerabilities, and enshrine assumptions required by software mitigations in clear architectural requirements. Despite specifying clear semantics, our contracts avoid limiting performance; the benchmark results for our prototype enforcements in RiscyOO incurred only 1.2% performance overhead. Crucially, we have also demonstrated that these contracts can be verified with microarchitectural tests.

Speculation contracts are particularly necessary for security architectural extensions, such as x86 SGX, CET, MPK, and SEV; ARM's MTE and PAC; and upcoming extensions such as CHERI [33].

We hope this work inspires further exploration of the trade-off between architectural speculation constraints and software mitigations. Cross-layer design and optimization will require expertise from multiple fields, as well as thorough evaluation of each design point. Architectural speculation contracts will be a key tool leveraged by future software countermeasures. These contracts will allow software to provide verifiable transient-vulnerability safety without expensive and conservative software constructs caused by a lack of architectural specification.

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium*, August 2018.

[2] P. Kocher, J. Horn, A. Fogh, 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 *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2019, pp. 1–19. [Online]. Available: https://doi.org/10.1109/SP.2019.00002

[3] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "Spectre returns! Speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies, WOOT 2018*, C. Rossow and Y. Younan, Eds. USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[4] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, January 2018, pp. 2109–2122. [Online]. Available: https://doi.org/10.1145/3243734.3243761

[5] J. Horn, "speculative execution, variant 4: speculative store bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, February 2018.

[6] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, pp. 719–732.

[7] "Vulnerability of speculative processors to cache timing side-channel mechanism," https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability, June 2022.

[8] C. Carruth, "Speculative load hardening," https://llvm.org/docs/SpeculativeLoadHardening.html, 2018.

[9] J. Wikner and K. Razavi, "RETBLEED: Arbitrary speculative code execution with return instructions," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3825–3842.

[10] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2021.

[11] "SPEC CPU 2006," https://www.spec.org/cpu2006/, January 2018.

[12] S. Ainsworth and T. M. Jones, "MuonTrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, pp. 132–144. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00022

[13] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (STT) a comprehensive protection for speculatively accessed data," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, October 2019, pp. 954–968.

[14] R. Choudhary, J. Yu, C. Fletcher, and A. Morrison, "Speculative privacy tracking (SPT): Leaking information from speculative execution without compromising privacy," ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 607–622. [Online]. Available: https://doi.org/10.1145/3466752.3480068

[15] H. Jin, Z. He, and W. Qiang, "SpecTerminator: Blocking speculative side channels based on instruction classes on RISC-V," *ACM Trans. Archit. Code Optim.*, nov 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3566053

[16] "Intel analysis of speculative execution side channels," Intel Corporation, Tech. Rep., January 2018. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/intel-analysis-of-speculative-execution-side-channels-paper.html

[17] Arm Limited, "Cache speculation side-channels," Tech. Rep. 2.5, June 2020. [Online]. Available: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability

[18] "Arm v8.5-A/v9 CPU updates," May 2021. [Online]. Available: https://developer.arm.com/documentation/102822/0105

[19] D. Mosberger, "Memory Consistency Models," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 1, pp. 18–26, 1993.

[20] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: Testing black-box CPUs against speculation contracts," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 226–239. [Online]. Available: https://doi.org/10.1145/3503222.3507729

[21] Arm Limited, "Straight-line speculation," Tech. Rep. 1.0, June 2020. [Online]. Available: https://developer.arm.com/support/arm-security-updates/speculative-processorvulnerability/downloads/straight-line-speculation

[22] Intel Corporation, "Retpoline: A branch target injection mitigation," Tech. Rep., June 2018. [Online]. Available: https://software.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf

[23] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, August 2019, pp. 249–266.

[24] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[25] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU register state using microarchitectural side-channels," *CoRR*, vol. abs/1806.07480, 2018. [Online]. Available: http://arxiv.org/abs/1806.07480

[26] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. V. Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant CPUs," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 769–784. [Online]. Available: https://doi.org/10.1145/3319535.3363219

[27] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2019, pp. 88–105. [Online]. Available: https://doi.org/10.1109/SP.2019.00087

[28] S. Zhang, A. Wright, T. Bourgeat, and Arvind, "Composable building blocks to open up processor design," in *51st Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO)*. IEEE Computer Society, 2018, pp. 68–81. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00015

[29] F. A. Fuchs, J. Woodruff, S. W. Moore, P. G. Neumann, and R. N. Watson, "Developing a test suite for transient-execution attacks on RISC-V and CHERI-RISC-V," in *Workshop on Computer Architecture Research with RISC-V*, 2021.

[30] F. A. Fuchs, "Analysis of Transient-Execution Attacks on the out-of-order CHERI-RISC-V Microprocessor Toooba," KTH Royal Institute of Technology, Tech. Rep., 2021.

[31] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu, "IntroSpectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities," in *48th Annual International Symposium on Computer Architecture (ISCA)*, June 2021, pp. 874–887.

[32] A. Joannou, P. Rugg, J. Woodruff, F. A. Fuchs, M. van der Maas, M. Naylor, M. Roe, R. N. M. Watson, P. G. Neumann, and S. W. Moore, "Randomized testing of RISC-V CPUs using direct instruction injection," *IEEE Design & Test*, 2023.

[33] R. Grisenthwaite, G. Barnes, R. N. M. Watson, S. W. Moore, P. Sewell, and J. Woodruff, "The Arm Morello evaluation platform – validating CHERI-based security in a high-performance system," *IEEE Micro*, vol. 43, no. 3, pp. 50–57, 2023.