CoatCheckMate: Verification of Microarchitectural Defenses

Paul Pok Hym Ng
ppng2@illinois.edu
University of Illinois at Urbana-Champaign

Shan Bai
shanbai2@illinois.edu
University of Illinois at Urbana-Champaign

ABSTRACT

Over the past few years hardware side-channel attacks have received much attention since the release and discovery of Spectre and Meltdown. Since then there have been many different defeneses that have been released. However due to the complexity of any given production microarchitecture, it is prohibitively hard to provide security guarantees via testing. In CheckMate [4] it was shown that given an abstraction of an exploit, it is then possible to check via bounded model checking if a microarchitecture is vulnerable. With this in hand, naturally the following question arises, "How do we verify that a defense always defends properly?". In this paper we aim to solve this problem by extending the CoatCheck [6]. By extending CoatCheck and CheckMate we have designed a system that is able to verify soundly that a defense works 100% given a microarchitecture.

KEYWORDS

Hardware description language, Formal verification, Memory Consistency Model, Theorem Proving, Side-Channels Attacks, Automated Verification

ACM Reference Format:
Paul Pok Hym Ng and Shan Bai. 2019. CoatCheckMate: Verification of Microarchitectural Defenses. In *CS598CLF '19: Secure Processor Design, October - December, 2019, Champaign, IL.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Ever since Meltdown and Spectre have been announced, hardware based side-channel attacks have been a very active field. Of major note is the number of speculation based attacks that have appeared over the past few years. There has been a variety of side channel attacks targeting different implementation specific behaviors with measurable states as proposed in [3, 7–10]. Therefore, proper verification tool that could detect possible leakage in the microarchitecture is critical. Meltdown and Spectre abuse properties of speculative execution which rely on memory structures such as the cache. In doing so, they make use of memory structures such as the cache. More specifically, they make use of `FLUSH+RELOAD` which allows an attacker to leak sensitive information stored in cache structures as a result of speculative execution.

CS598CLF '19, October - December, 2019, Champaign, IL

ACM ISBN 123-4-5678-9012-3/45/67…$69.69
https://doi.org/10.1145/1122445.1122456

However, due to the fact that these leakages occur at the microarchitectural level, there is no resulting physical observable state to detect an attack. One way to model microarchitectural level is to use a memory consistency model. The Check suite of tools developed at Princeton use a concept known as μhb (microarchitecturally happens before) graphs. Such graphs have their origin in earlier memory consistency model (MCM) work and depict the movement of data and relations between instructions/data throughout the pipeline.

CheckMate takes μhb graphs further and adapts this for security purposes. CheckMate generalizes the idea of a litmus test to something known as an **exploit pattern**. An exploit pattern is the most general way of representing a list of instructions that an exploit contains. These replace the classical litmus tests that are used in MCMs. In CoatCheckMate, we take this same idea and apply it CoatCheck to soundly check if a defense works by making the following contributions:

- We build a classical 5-stage microarchitecture model without speculation on a multi-threaded core and verify its susceptibility to the `FLUSH+RELOAD` attack.
- To model `FLUSH+RELOAD`, we implement support for a Flush operation.
- We implement a simple defense by disabling shared read only memory, and verify it succeeds in defending against the attack
- We demonstrate the potential of modeling microarchitecture and formaly verifying it without the use of bounded model checking by using an axiomatically defined model

2 BACKGROUND

2.1 μhb Example

We begin with an example of a microarchitecturally happens before graph. This example is taken from PipeCheck [5]. In this case we have two cores, each with two instructions. Core 0 writes a 1 to two

Core 0	Core 1
(i1) [x] ← 1	(i3) r1 ← [y]
(i2) [y] ← 1	(i4) r2 ← [x]
Under TSO: Forbid? r1=1, r2=0	

Figure 1: Litmus Test Code [5]

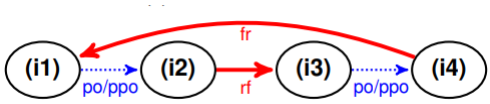


Figure 2: Litmus Test Program Order [5]

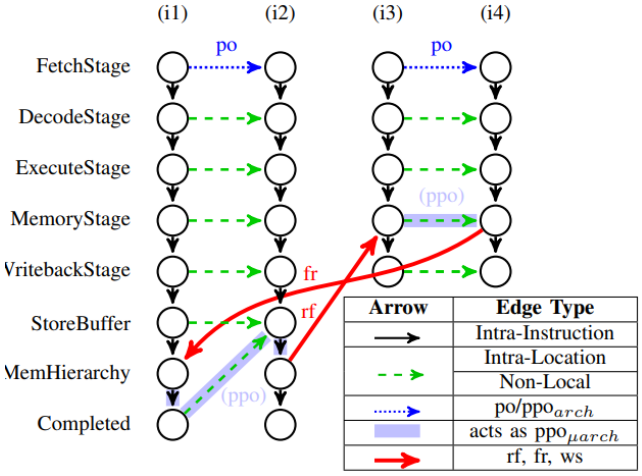


Figure 3: Litmus Test $u$hb Graph [5]

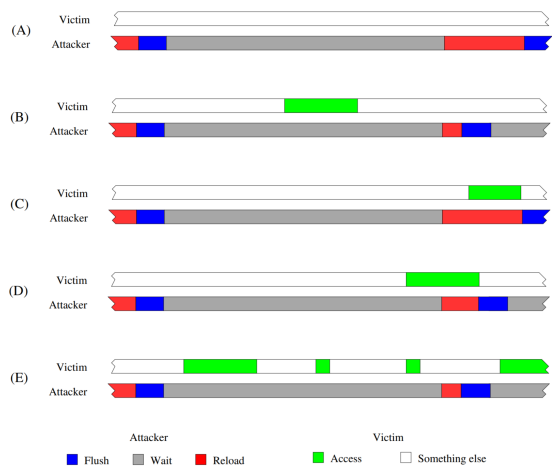


Figure 4: Timing of FLUSH+RELOAD. (A) No Victim Access (B) With Victim Access (C) Victim Access Overlap (D) Partial Overlap (E) Multiple Victim Accesses [12]

different addresses x, y and core 1 reads addresses x, y. We wish to check if it is possible for register 1 and 2 (r_1, r_2) can have values 1 and 0 respectively. As we can see from Figure 2 this execution path creates a cyclic graph. This means that this particular execution with these specific results are impossible. In Figure 3 we can see the same cyclic behavior. In the case of CoatCheckMate this would mean that a specific exploit pattern is impossible.

In a μhb graph there are two groups of edges. Firstly there are static edges that are generated by the program itself. These are edges such as program order and fence edges. On the other hand we have **observed edges** of which there are 3 types. These were first highlighted by Alglave [2] who described them as follows. For an edge (s, d)

(1) A "reads from" (rf) edge is one where d reads from s. Therefore s must happen before d.
(2) A "from reads" (fr) edge is oen where s reads a value from a write that occurred before d in the set of ws edges.
(3) A "write serialization/coherence order" (ws) edge is one where s comes before d in the assumed program ordering from the viewpoint of the memory hierarchy.

2.2 Why Build off of CoatCheck Rather than CheckMate?

CheckMate is written in Alloy which is designed for bounded model checking. As a result to ensure that such a model-finding problem is decidable, Alloy restricts the size of the model to a user-defined finite number of objects. The designers argued that most bugs (especially pathological ones) are found by small set of test inputs and as such this is "good enough" [1]. This is acknowledged in [4] in the section discussing "Constraining Solutions" with regards to constraining cache coherence activity.

For CheckMate this is good enough as the goal is only to show that a specific exploit pattern **can** occur. In CoatCheckMate this guarantee is not enough. Therefore we must build off of CoatCheck which is written in Coq, an interactive theorem prover. This gives us a soundness guarantee that given an exploit pattern and a microarchitecture we will always get a definitive answer stating whether an exploit pattern can execute or not.

2.3 FLUSH+RELOAD A Refresher

Here we discuss what FLUSH+RELOAD [12] is. FLUSH+RELOAD is a variant of PRIME+PROBE with the difference being that FLUSH+RELOAD requires shared memory pages between a victim and attacker. An attack contains 3 phases. In the first phase, the attacker flushes a monitored line from memory. The second phase entails attacker then waits for the victim to access that specific line of memory. Finally in the third phase, the attacker reloads the line measuring the time required to load it. This enables the classic RSA exponentiation algorithm to be exploited where the attacker can discover the number of 1s in an encryption key.

3 MODELING A CACHE

As we are modeling a FLUSH+RELOAD attack we must model some form of cache-like structure. We use something known as a Value in Cache Line (ViCl) abstraction. This was first introduced in CCICheck [11]. A ViCl is a 4-tuple

$$(cacheId, address, dataValue, generationId)$$

where

(1) $cacheId$: The cache which a ViCl is tied to
(2) $address$: The address of which the data value resides
(3) $dataValue$: The value in memory $generationId$: A "timestamp" of sorts representing when a ViCl was created

A ViCl's lifecycle is then described by 3 node types. ViClCreate and ViClExpire represent points in time where a ViCl begins and stops serving a data value in cache. ViClDowngrade represents when a cache line goes from an exclusive to shared state. In the case of CoatCheckMate and more specifically the FLUSH+RELOAD attack we

Thread 0	Thread 1
(i1) [x] ← 1	(i3) r1 ← [x]
(i2) [x] ← 2	(i4) r2 ← [x]
In TSO: r1=2, r2=2 Allowed	

Figure 5: Litmus Test `co-mp` [11]

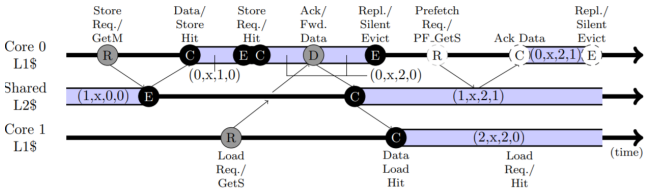


Figure 6: `co-mp` ViCl Lifecycle [11]

do not take into account ViClDowngrade events. We do not need to do so because the cached data in question is already in a shared state as a requirement.

Below we show another litmus test and its corresponding ViCl events [11]. (i1) misses in its cache and thus is required to retrieve the value from the shared L2 cache causing the L2 ViCl to expire. Afterwards, due to the update from (i2), the ViClCreate event from (i1) expires triggering ViClExpire (i1) and ViClCreate (i2). When (i3) executes some time later, it must retrieve the data from Core 0 instead of the L2 as it is stale. This triggers (i2)'s ViCl to expire and we must add a ViClDowngrade event.

The good thing about the ViCl abstraction is that one can use other predicates and supporting structures to determine whether a two threads or processors have virtual addresses that resolve to the same physical address. This is something that we use to great effect in CoatCheckMate to represent shared read only memory.

4 FLUSH+RELOAD CASE STUDY

Due to the lack of time we elected to build our microarchitecture from the basic 5-stage pipeline model. However this meant that we would have to add a cache abstraction to a 5-stage pipeline in order to model `FLUSH+RELOAD`. To do this there were several major things that we needed to augment the COATCheck codebase with.

(1) Create a flush instruction
(2) Predicates relating to flush instructions
(3) Enable the parser to parse a flush instruction
(4) Create a way to output flush instructions in the output graph
(5) Predicates relating to cacheable memory
(6) Modify memory addresses to include an extra flag marking whether it is cacheable
(7) Predicates relating to caches
(8) Modify 5-stage pipeline with cache axioms and flush axioms
(9) For `FLUSH+RELOAD` specify the pattern and axioms relating to it and shared read only memory
(10) Create a litmus test representing the FLUSH+RELOAD pattern

With these changes CoatCheckMate now has the capability to prove that `FLUSH+RELOAD` can run on a vulnerable microarchitecture and an added defense will guard against `FLUSH+RELOAD`.

4.1 Defining An Attacker and Victim

As discussed in the FLUSH+RELOAD refresher section, an attack requires both an attacker process and a victim process. We do not do this. While it would be relatively simple to do so by modifying CoatCheck to input one more argument in the litmus test marking whether a process is an attacker or victim, we opted (due to lack of time) to use two different threads to represent an attacker and a victim.

However, this limits us to single threaded programs on a single core. For now, we decided that as a proof of concept this would fall into the category of future work as our abstraction of processes to threads is sufficient for our proof of concept.

4.2 Specifying the Microarchitecture

COATCheck comes pre-supplied with a simple 5-stage microarchitecture. Therefore we only needed to specify cache, flush, and `FLUSH+RELOAD` related axioms and predicates. Due to the differences between Alloy (which CheckMate was written in) and Coq and more specifically the DSL that COATCheck uses, this was quite a task.

In order to support caches, we needed to add two new stages to the pipe line, ViClCreate and ViClExpire. These live between the `StoreBuffer` and `MemoryHierarchy` stage.

```
StageName 0 "Fetch".
...
StageName 5 "StoreBuffer".
StageName 6 "ViCLCreate".
StageName 7 "ViCLExpire".
StageName 8 "MemoryHierarchy".
StageName 9 "Complete".
```

We then have to add various cache related axioms. Below is the axiom for `L1ViClNoDups`. This specific axiom can be seen in action in Figure 6 where (i1) must expire in order for (i2) to create its own ViClCreate event.

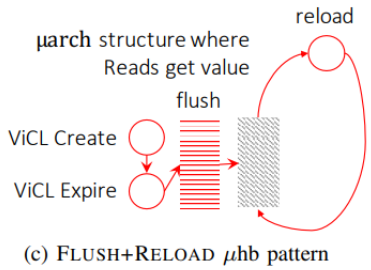
```
Axiom "L1ViCLNoDups":
  forall microop "e",
  forall microop "ee",
  (
    (IsAnyRead e \/ IsAnyWrite e)
    /\ (IsAnyWrite ee \/ IsAnyRead ee) =>
    (
      (
        ~(SameMicroop e ee) /\ SameCore 0 c
        /\ (SameVirtualTag e ee \/ SamePhysicalAddress e ee)
        /\ NodesExist[(e, ViCLCreate); (ee, ViCLCreate)]
      )
    ) =>
```

```
    EdgeExists((e, ViCLExpire), (ee, ViCLCreate), "")
    \/
    EdgeExists((ee, ViCLExpire), (e, ViCLCreate), "")
  ).
```

The above axiom can be read as follows. For any 2 different memory events (read or write) that resolve to the same physical address or have the same cache index, we will have an edge between one instruction's ViClExpire and the other instruction's ViClCreate node.

To specify FLUSH+RELOAD we took a different approach than CheckMate. Rather than specifying a set of predicates that must be true when calling the Alloy model checker we added FLUSH+RELOAD as an axiom in the following way,

```
Axiom "flush_reload":
  exists microop "f",
  exists microop "i0",
  exists microop "i2",
  exists microop "i3",
  (
    IsFlush f /\ IsAnyRead i2
    /\ IsAnyRead i0 /\ IsAnyRead i3
    /\ ~SameMicroop f i2 /\ ~SameMicroop i2 i3
    /\ ~SameMicroop i3 i0
    /\ OnThread 2 f /\ OnThread 2 i2
    /\ OnThread 2 i3 /\ OnThread 1 i0
    /\ IsCacheable i0 /\ IsCacheable i2
    /\ IsCacheable i3
    /\ NodesExist[(i2, ViCLCreate); (i2, ViCLExpire)]
    /\ EdgesExist[
    ((i2, ViCLCreate), (i2, ViCLExpire), "FLRE", "orange");
    ((i2, ViCLExpire), (f, Execute), "FLRE", "orange");
    ((f, Complete), (i0, Fetch), "FLRE", "orange");
    ((i0, ViCLCreate), (i3, Execute), "FLRE", "orange");
    ((i3, Execute), (i0, ViCLExpire), "FLRE", "orange")
    ]
  ).
```



(c) FLUSH+RELOAD μhb pattern

Figure 7: Basic pattern [4]

If we look at Figure 7 these are the edges that define FLUSH+RELOAD pattern. With this in mind our approach was to define all the relations between the 4 micro-ops that the FLUSH+RELOAD pattern has

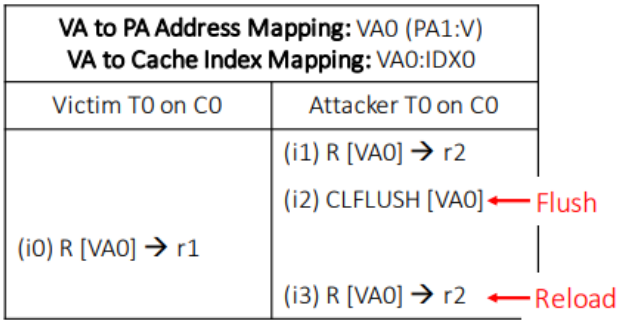


Figure 8: Instructions for Flush Reload attack [4]

and then conjunct that with fact that these minimal edges can be added mean that a FLUSH+RELOAD can occur.

One final important axiom of note is the one regarding shared read only memory.

```
Axiom "shared_ro_mem":
  exists microop "i",
  exists microop "i'",(
    OnThread 1 i /\ OnThread 2 i'
    /\ IsAnyRead i /\ IsAnyRead i'
    /\ SameVirtualTag i i' /\ ~SameMicroop i i'
    /\ SamePhysicalAddress i i'
  ).
```

This can be read as follows. If we have two reads that are not the same micro-op, on two different threads, and have the same virtual tag, they must also have the same physical address.

4.3 Designing the Litmus Test

The litmus test was simple to design. We essentially followed the same instructions described in [4]. The only discrepancy between this and our litmus test is that this figure shows that both attacker and victim are on the same thread. This gave us the following litmus test.

```
Alternative
0 0 1 0 (Read normal (VA 0 0 1) (PA 1 0) (Data 0))
1 0 2 0 (Read normal (VA 0 0 1) (PA 1 0) (Data 0))
2 0 2 0 (Flush normal (VA 0 0 1) )
3 0 2 0 (Read normal (VA 0 0 1) (PA 1 0) (Data 0))
Relationship po 1 0 -> 2 0
Relationship po 2 0 -> 0 0
Relationship po 2 0 -> 3 0
Relationship po 2 0 -> 0 0
Relationship flush 2 0 -> 3 0
Relationship flush 2 0 -> 0 0
Permitted
```

As you can see, we have three read instructions, all on the same physical and virtual addresses. In addition, the flush instruction

performed by the attacker will help reveal the related address information on the victim side. This is the exact set of instructions from Figure 8.

The format is as follows. An instruction has the format shown below.

```
<globalID> <coreID> <threadID> <intraInstID> <opcode>
```

Relationships between instructions have the format shown below between *inst*0 and *inst*1.

```
Relationship <name> <globalID0> <globalID1>
```

The `<name>` field is user customizable and is used for string matching when using the `HasDependency` predicate. The only change from CoatCheck is that our virtual address has one more field. The third entry of `VA` represents whether the memory address is cacheable. For example, po represents program order and defines which instruction must happens before the other.

5 THE DEFENSE

As a proof of concept we wished to keep things simple. With that in mind, we chose the simplest solution. Turn off shared read only memory. The axiom to that is shown below.

```
Axiom "no_shared_ro_mem":
  exists microop "i",
  exists microop "i'",(
    (OnThread 1 i /\ OnThread 2 i'
    /\ IsAnyRead i /\ IsAnyRead i'
    /\ SameVirtualTag i i' /\ ~SameMicroop i i'
    /\ ~SamePhysicalAddress i i')
  ).
```

Here the only difference from the shared read only memory axiom is that even if you have the same virtual tag you cannot resolve to he same physical address. In doing so, we have defended against the `FLUSH+RELOAD` exploit.

6 EVALUATION AND RESULT

As discussed above, in our work we pick `FLUSH+RELOAD` as our attack and use a simple defense that turns off shared read only memory. As shown in Figure 7, the path in orange shows the minimal exploit pattern for the attack. To enable the attack, we explicitly allow shared read only memory through an axiom that allow different read instruction to access the same physical address. Figure 9 shows our generated μhb graph with the attack enabled. As we were able to generate a graph, CoatCheckMate has checked that there are no cyclic paths and therefore observable. The orange path matches Figure 7 which demonstrates the correctness of our attack.

To verify the correctness of our defence, we replace the above axiom with one that disable access to the same physical address. As expected, we get a graph as shown in 10 where such an execution is trivially evaluated false. This means the attack instruction path is not possible with the shared read only memory turned off. These two figures together demonstrate that CoatCheckMate is able to prove a defense will work thus giving our proof soundness.

7 CHALLENGES

There were several challenges we encountered during the course of development. Firstly, this was our first experience with the languages used in COATcheck and CheckMate (Coq and Alloy). Therefore a significant portion of our time spent on CoatCheckMate was spent learning these languages in order to understand and augment the code base.

Due to the differences in the DSL used in COATCheck and Alloy, defining certain axioms was difficult. For example, Alloy has a `disj` keyword which allows one to state that two variables (micro-ops) must be different. However in COATCheck this is achieved via the `SameMicroop` predicate. Another difference was the fact that `DefineMacro` in COATCheck's DSL and `pred` in Alloy behave differently. In Alloy predicates can take arguments which significantly simplifies the use of predicates. On the other hand `DefineMacro` does not support this feature. Therefore if you have two different axioms which need to use the same macro (and have different micro-op names), you either need to define two different macros or just inline the macro yourself.

Secondly, correctly defining a microarchitecture is hard. Just because one litmus test works does not mean the next one will. Due to time constraints, we did not have the luxury of verifying multiple litmus tests and have just implemented the `FLUSH+RELOAD` attack and its defense. This was further hampered by the fact that we added one extra field to our virtual address meaning that we would have had to modify every single existing litmus test. We only verified the correctness of one other litmus test (`sb.test`).

Lastly, the codebase for COATCheck itself is outdated and not supported in the newest version of Coq (8.10.2 at the time of writing) and OCaml (4.09.0 at the time of writing). While this in itself is not a problem due to the ability to pin to a specific version in opam, if COATCheck and by extension CoatCheckMate were to be deployed in an actual industry environment it would be best to fix all the compatibility issues.

8 LIMITATIONS AND FUTURE WORK

As hinted earlier our model, though correct axiomatically, has limitations. First and foremost, unlike CheckMate, CoatCheckMate cannot automatically generate new attacks. Currently, the user has to manually define new axioms and supporting operations in order to verify exploit patterns. This functionality would be crucial not only for usability but help us better evaluate an architecture. Another limitation is that it is hard to verify the correctness of the full architecture. This could be improved by verifying the existing litmus test suite on our augmented five stage microarchitecture. A more general limitation of the COATCheck codebase is that the SMT solver used is quite primitive. In the course of implementing CoatCheckMate we incorrectly added an axiom with too many conjunctions and variables leading to a prohibitively long runtime (> 1 hour) which then promptly crashed the graph generation tool, neato. While improving upon this is currently outside our purview, a better SMT solver would be desirable if we were to try and prove more complex exploit patterns that are longer than a handful of

instructions.

With regards to our proof of concept using FLUSH+RELOAD we definitely took a few liberties. As previously mentioned instead of using a process abstraction, we ended up separating attacker and victim via threadIDs. This itself limits both victim and attacker to single threaded processes running on a single core. In order to make this a more general verification we would have to implement this abstraction. It should not be hard to do so as we would only need to add a predicate called IsAttacker and a new boolean field in the litmus test which would mark which instructions belong to the attacker and which belong to the victim. This would then allow us to avoid a victim benignly probing their own cache lines in PRIME+PROBE and be mistaken for an attacker.

9 CONCLUSION

CoatCheckMate is a proof of concept that allows a user to axiomatically define exploit patterns, microarchitectures, defenses and verify that it correctly defends against an attack. Due to the complexity of microarchitecture designs, this gives chip designers an automated way of guaranteeing the safety of the design.

While CoatCheckMate is only a proof of concept, it shows potential for future development. We hope that this work is good jumping off point for integrating more features into the suite of Check programs.

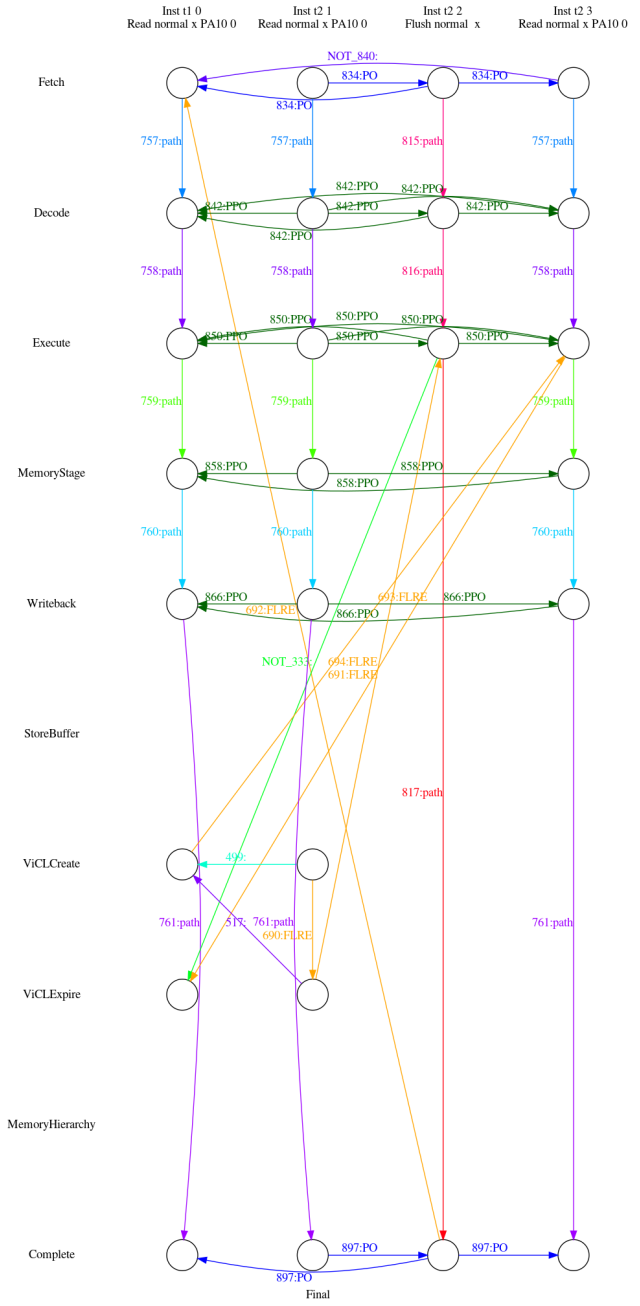


Figure 9: μhb graph for Flush + Reload attack

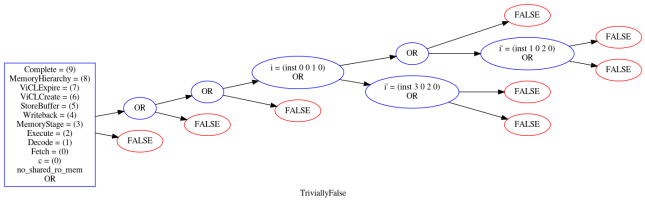


Figure 10: Trivially false after the defense is enabled

REFERENCES

[1] Sarfraz Khurshid Darko Marinov Alexandr Andoni, Dumitru Daniliuc. [n. d.]. Evaluating the "Small Scope Hypothesis".

[2] Jade Alglave. [n. d.]. A formal hierarchy of weak memory models.

[3] Michael Backes, Markus Dürmuth, Sebastian Gerling, Manfred Pinkal, and Caroline Sporleder. 2010. Acoustic Side-channel Attacks on Printers. In *Proceedings of the 19th USENIX Conference on Security (USENIX Security'10)*. USENIX Association, Berkeley, CA, USA, 20–20. http://dl.acm.org/citation.cfm?id=1929820.1929847

[4] Margaret Martonosi Caroline Trippel, Daniel Lustig. [n. d.]. CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests.

[5] Margaret Martonosi Daniel Lustig, Michael Pellauer. [n. d.]. PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models.

[6] Margaret Martonosi Abhishek Bhattacharjee Daniel Lustig, Geet Sethi. [n. d.]. COATCheck: Verifying Memory Ordering at the Hardware-OS Interface.

[7] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. 38–55. https://doi.org/10.1109/SP.2016.11

[8] David Gullasch, Endre Bangerter, and Stephan Krenn. 2010. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. *2011 IEEE Symposium on Security and Privacy* (2010), 490–505.

[9] D. Harnik, B. Pinkas, and A. Shulman-Peleg. 2010. Side Channels in Cloud Services: Deduplication in Cloud Storage. *IEEE Security Privacy* 8, 6 (Nov 2010), 40–47. https://doi.org/10.1109/MSP.2010.187

[10] N. Homma, T. Aoki, and A. Satoh. 2010. Electromagnetic information leakage for side-channel analysis of cryptographic modules. In *2010 IEEE International Symposium on Electromagnetic Compatibility*. 97–102. https://doi.org/10.1109/ISEMC.2010.5711254

[11] Michael Pellauer Margaret Martonosi Yatin A. Manerkar, Daniel Lustig. [n. d.]. CCICheck: Using uhb Graphs to Verify the Choherence-Consistency Interface.

[12] Katrina Falkner Yuval Yarom. [n. d.]. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.