Εθνικο Μετσοβιο Πολυτεχνειο
Τμημα Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστήριο Υπολογιστικών Συστημάτων

Μελέτη και αξιολόγηση μηχανισμών προστασίας από επιθέσεις παράπλευρων καναλιών υποθετικής εκτέλεσης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ
του

ΘΕΟΔΩΡΟΥ ΤΡΟΧΑΤΟΥ

Επιβλέπων: Διονύσιος Πνευματικάτος
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2021

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΉΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Μελέτη και αξιολόγηση μηχανισμών προστασίας από επιθέσεις παράπλευρων καναλιών υποθετικής εκτέλεσης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΘΕΟΔΩΡΟΥ ΤΡΟΧΑΤΟΥ

Επιβλέπων: Διονύσιος Πνευματικάτος
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 5η Ιουλίου 2021

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Νεκτάριος Κοζύρης	Διονύσιος Πνευματικάτος	Γεώργιος Γκούμας
Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Αν.Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2021

Εθνικο Μετσοβιο Πολυτεχνειο

Τμημα Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστήριο Υπολογιστικων Συστηματων

(Υπογραφή)

.....

Θεόδωρος Τροχάτος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

National Technical University of Athens

School of Electrical & Computer Engineering
Division of Communication, Electronic and Information Engineering

Survey of Transient Execution Attacks and Mitigation Mechanisms

Theodoros Trochatos

Supervisor: Dionisios Pnevmatikatos
Professor ECE NTUA

Computing Systems Laboratory

Athens, July 2021

Περίληψη

Η υποθετική εκτέλεση εντολών (speculative execution) είναι μια τεχνική που χρησιμοποιείται εδώ και αρκετά χρόνια στη σχεδίαση των σύγχρονων επεξεργαστών με σκοπό την αύξηση της απόδοσής τους. Αυτή η τεχνική δεν φαινόταν να δημιουργεί κάποια θέματα ασφαλείας στους επεξεργαστές, έως ότου το 2018 εμφανίστηκαν οι επιθέσεις τύπου Spectre και Meltdown. Οι επιθέσεις αυτές είναι επικίνδυνες και δύσκολα αντιμετωπίσιμες, καθώς εκμεταλεύονται δυο θεμελιώδεις τεχνικές για την αύξηση της απόδοσης του επεξεργαστή: την υποθετική εκτέλεση και την εκτέλεση εκτός σειράς εντολών. Για αυτό το λόγο, οι μηχανισμοί άμυνας έναντι αυτών των επιθέσεων που είναι βασισμένοι στο λογισμικό κρίνονται ανεπαρκείς για δυο λόγους. Πρώτον, επιφέρουν μέγαλη επιβράδυνση στην απόδοση του επεξεργαστή. Δεύτερον, δεν παρέχουν καθολική προστασία από όλες τις παραλλαγές αυτών των επιθέσεων, καθώς οι επιθέσεις αυτές εκμεταλλεύονται κυρίως την υποθετική εκτέλεση εντολών, γεγονός που αφορά το πώς είναι δομημένος ο επεξεργαστής. Έτσι, τα τελευταία χρόνια αναπτύχθηκαν μηχανισμοί άμυνας βασισμένοι στο υλικό, παρέχοντας γενικότερη προστασία και διατηρώντας υψηλή απόδοση. Σε αυτήν τη διπλωματική εργασία κάνουμε μια λεπτομερή καταγραφή των διαφόρων επιθέσεων τύπου Spectre και Meltdown, πώς λειτουργούν και πώς καταφέρνουν να υποκλέψουν ευαίσθητες πληροφορίες, αλλά και των μηχανισμών άμυνας βασισμένων στο υλικό. Συγκεκριμένα, κατηγοριοπούμε τις τεχνικές άμυνας σε τρείς κύριες κατηγορίες: σε τεχνικές απόκρυψης, σε τεχνικές καθυστέρησης και σε τεχνικές αναίρεσης. Παράλληλα, αξιολογούμε τα πλεονεκτήματα και τα μειονεκτήματα των μηχανισμών της καθε κατηγορίας. Τέλος, πραγματοποιούμε αναπαραγωγή των πειραματικών αποτελεσμάτων από ένα μηχανισμό άμυνας της κάθε κατηγορίας μέσω του προσομοιωτή gem5 και των μετροπρογραμμάτων της σουίτας SPEC CPU2006. Η πειραματική μας αξιολόγηση δείχνει ότι οι μηχανισμοί που ανήκουν στις κατηγορίες των τεχνικών απόκρυψης και αναίρεσης προσφέρουν την καλύτερες επιδόσεις, όμως καλύπτουν παράπλευρα κανάλια που σχετίζονται μόνο με την Data-Cache. Σε αντίθεση, οι μηχανισμοί που ανήκουν στις τεχνικές καθυστέρησης, αν και συνήθως εισάγουν περισσότερο overhead, παρέχουν καθολική προστασία για κάθε παράπλευρο κανάλι που μπορεί να εκμεταλλευτεί ένας κακόβουλος χρήστης.

Λέξεις Κλειδιά

Υποθετική εκτέλεση, Εκτέλεση εκτός σειράς, Κρυφή μνήμη, Παράπλευρα κανάλια, Επιθέσεις παράπλευρων καναλιών, Μηχανισμοί άμυνας

Abstract

Speculative execution is one the most broadly known techniques used for improving the performance of processors. Until recently, speculative execution has not any security implications. However, the recent disclosure of transient execution attacks in January 2018 has called the security protections of processors into question as they can cause critical data leakage. Namely, Spectre and Meltdown are threats that are challenging to defeat, because they exploit two fundamental concepts of modern processors that boost performance and efficiency which are completely independent of the Operating System they running: Speculative execution and Out-of-Order execution. On top of that, proposed software-based mitigation solutions are considered to be inadequate for two main reasons. First, software-based defenses suffer from excessive high overhead. Second, software-based fail to provide a catholic and comprehensive protection against every cache-based side channel attack. Therefore, the severity of the transient execution attacks has motivated computer architects in both industry and academia to rethink the design of the processors and to propose hardware defenses. In this diploma thesis we describe in detail how Spectre and Meltdown type attacks work. In addition, we analyze and summarize proposed hardware mitigation mechanisms and categorize them into three main approaches: (i) hiding-based, (ii) delaying-based, and (iii) undo-based solutions. Finally, we replicate and evaluate the statistics results of one mitigation mechanism of each category by using the gem5 simulator and the SPEC CPU2006 benchmarks, and discuss the possible trade-offs among the mechanisms. Our evaluation shows that hiding-based and undo-based solutions incur usually lower performance overhead, but they protect only data caches. In contrast, delaying-based solutions although they incur usually higher performance overhead they achieve catholic protection of any covert channel.

Keywords

Speculative execution, Out-of-Order execution, Cache, Side channels, Spectre, Meltdown, Mitigation mechanisms

Ευχαριστίες

Με αυτήν τη διπλωματική ολοκληρώνεται ένα μακρύ, δύσκολο αλλά και συνάμα πολύ όμορφο ταξίδι 6 ετών στο Εθνικό Μετσόβειο Πολυτεχνείο. Στη διάρκεια αυτού του ταξιδιού απόκτησα πολλές γνώσεις και εμπειρίες, έκανα φιλίες ζωής, καρδιοχτύπησα. Όλα αυτά με έκαναν έναν πιο ολοκληρωμένο άνθρωπο.

Θα ήθελα να ευχαριστήσω τον Καθηγήτη Διονύσιο Πνευματικάτο για την ευκαιρία που μου έδωσε να εκπονήσω αυτή τη διπλωματική εργασία στο Εργαστήριο Υπολογιστικών Συστημάτων. Μέσω αυτής της διπλωματικής εισήλθα σε μονοπάτια της σύγχρονης έρευνας. Επιπλέον, θα ήθελα να ευχαριστήσω τον Δόκτορα Βασίλη Καρακώστα για την εξαιρετική μας συνεργασία στα πλαίσια της διπλωματικής εργασίας, που αν και είχε φορτωμένο πρόγραμμα όλη τη χρονιά ήταν πάντα εκεί για να με συμβουλεύει υπομονετικά. Επίσης, θα ήθελα να τους ευχαριστήσω για τη βοήθεια και για τις συμβουλές που μου παρείχαν κατά τη διάρκεια των αιτήσεών μου για διδακτορικό στην Αμερική. Η συνεισφορά τους ήταν πολύτιμη και πηγαίνοντας βήμα βήμα σε κάθε στάδιο, κατάφερα κάτι που δεν το είχα ούτε ονειρευτεί.

Τέλος, θα ήταν παράλειψη να μην ευχαριστήσω τους γονείς μου, τους αφανείς ήρωες. Σε κάθε βήμα και σε κάθε προσπάθειά μου ήταν εκεί για να με στηρίζουν. Τους ευχαριστώ για την αγάπη τους αλλά και για την αδιάκοπη προσπάθειά τους να μου τονίζουν ότι στη γνώση και στην εκπαίδευση αξίζει να επενδύει κανείς για μια καλύτερη ζωή.

Θεόδωρος Τροχάτος,
Ιούλιος 2021

Περιεχόμενα

1 Εκτενής Περίληψη **14**

1.1 Θεωρητικό Υπόβαθρο 16

1.1.1 Εκτέλεση εντολών εκτός σειράς 16

1.1.2 Υποθετική εκτέλεση εντολών 16

1.2 Πώς λειτουργούν οι επιθέσεις; 17

1.2.1 Επιθέσεις Παράπλευρων καναλιών 17

1.2.2 Μεταβατικές Επιθέσεις Παράπλευρων Καναλιών 18

1.2.3 Αιτίες Μεταβατικής Εκτέλεσης 19

1.2.4 Disclosure Gadget 20

1.2.5 Ανακτώντας την ευαίσθητη πληροφορία 20

1.3 Πώς λειτουργεί το Meltdown 21

1.4 Πώς λειτουργεί το Spectre 21

1.5 Μηχανισμοί Άμυνας βασισμένοι στη Σχεδίαση Υλικού 22

1.5.1 Τεχνικές Απόκρυψης 22

1.5.2 Τεχνικές Καθυστέρησης 24

1.5.3 Τεχνικές Αναίρεσης 25

1.6 Μεθοδολογία 25

1.7 Πειραματική Αξιολόγηση Επίδοσης 26

2 Introduction **30**

2.1 Motivation 30

2.2 Approach 31

3 Background **33**

3.1 Out-of-Order Execution 33

3.2 Speculative Execution 33

4 How The Attackers Work? 35

4.1 Side Channel Attacks 35

4.2 Transient Execution Attacks 36

4.2.1 Causes of Transient Execution 37

4.2.2 Covert Channels 38

4.2.3 Disclosure Gadget 38

4.2.4 Recovering the secret 39

4.3 How Meltdown works 41

4.4 How Spectre works 41

4.5 Spectre & Meltdown Diferrencies 44

5 Hardware Mitigation Mechanisms 45

5.1 Hiding the side-effects of speculative execution until speculation is resolved 45

5.1.1 InvisiSpec 45

5.1.2 SafeSpec 49

5.1.3 Ghost Loads 51

5.1.4 Comparison Summary 53

5.2 Delaying speculative execution until speculation can be resolved 54

5.2.1 Speculative Taint Tracking (STT) 54

5.2.2 Delay-On-Miss with Value Prediction 57

5.2.3 NDA 59

5.2.4 DOLMA 59

5.3 Undoing the side-effects of speculative execution in the case of a mis-speculation: 62

5.4 Software Based Defenses 64

5.5 Other Mitigation Techniques 64

5.6 Summary 64

6 Methodology **66**

6.1 Simulation environment 66

7 Evaluation **69**

7.1 InvisiSpec 69

7.2 STT Results 71

7.3 DOLMA Results 72

7.4 CleanupSpec 74

7.5 Comparison 75

8 Conclusions **81**

9 Future Work **82**

1 Εκτενής Περίληψη

Στο πέρασμα των δεκαετιών, οι μηχανικοί υπολογιστών και ειδικότερα οι σχεδιαστές της αρχιτεκτονικής των υπολογιστών εργάζονταν σκληρά με σκοπό την επίτευξη της μέγιστης απόδοσης του επεξεργαστή διατηρώντας χαμηλά τις διαστάσεις του και το ενεργειακό κόστος. Η υποθετική εκτέλεση εντολών είναι μια θεμελιώδης τεχνική που χρησιμοποιείται κατά κόρον τις τελευταίες δεκαετίες με σκοπό την επίτευξη του παραπάνω δύσκολου στόχου. Παρόλο που αυτή η τεχνική ήταν ένα ισχυρό εργαλείο για την κατασκευή γρήγορων επεξεργαστών χωρίς να εμφανίζονται θέματα ασφάλειας, τα τελευταία χρόνια η ανάδειξη στο προσκήνιο των επιθέσεων που εκμεταλλεύονται την υποθετική εκτέλεση εντολών (Spectre [13] και Meltdown [15]) έχει θέσει σοβαρά ερωτήματα για την ασφάλεια των επεξεργαστών. Το γεγονός ότι εκμεταλλεύονται μια θεμελιώδη τεχνική που στηρίζονται οι επεξεργαστές και είναι εντελώς ανεξάρτητες του λειτουργικού συστήματος, τις καθιστά μια εξαιρετικά σημαντική απειλή. Ο λόγος είναι ότι αν και δεν αλλάζει την αρχιτεκτονική κατάσταση, η υποθετική εκτέλεση μπορεί να μεταβάλλει την μικροαρχιτεκτονική κατάσταση του επεξεργαστή. Παραδείγματος χάριν, κατά τη διάρκεια της υποθετικής εκτέλεσης των εντολών, οι προσβάσεις στην (cache) εκτελούνται κανονικά και αφήνονται κάποια ίχνη. Στη συνέχεια ένας κακόβουλος χρήστης ή λογισμικό με διάφορους τρόπους εκμεταλλέυεται τα παράπλευρα αποτελέσματα και τα ίχνη της υποθετικής εκτέλεσης και φτάνει στην υποκλοπή ευαίσθητων πληροφοριών.

Αν και προσφάτως έχουν δημιουργηθεί αρκέτες τεχνικές άμυνας βασισμένες στο λογισμικό (Software) ενάντια στις προαναφερθείσες επιθέσεις, η απόδοσή τους κρίνεται ανεπαρκής. Δυο είναι οι κύριοι λόγοι αυτής της αξιολόγησης. Ο πρώτος λόγος είναι ότι οι τεχνικές άμυνας σε επίπεδο λογισμικού έχουν πολύ μεγάλο κόστος παρουσιάζουντας πολύ μεγάλη επιβράδυνση στην απόδοση των επεξεργαστών (της τάξεως του 50% και μεγαλύτερη) καθιστώντας τους μη πρακτικούς. Ο δεύτερος λόγος είναι ότι, όπως προαναφέρθηκε οι επιθέσεις αυτές είναι ανεξάρτητες του λογισμικού. Αυτό σημαίνει ότι δεν υπάρχει ένας καθολικός τρόπος αντιμετώπισης τέτοιου είδους επιθέσεων μέσω τεχνικών άμυνας στο λογισμικό, αφού η υποθετική εκτέλεση εντολών είναι κάτι που αφορά τη λειτουργία του υλικού του επεξεργαστή.

Δεδομένου των παραπάνω, οι επιθέσεις τύπου Spectre και Meltdown έχουν κεντρίσει το ενδιαφέρον των μηχανικών υπολογιστών και τους έχει κάνει να στραφούν σε άλλου είδους λύσεις, πιο πρακτικές, παρέχοντας πιο καθολική

κάλυψη και μικρότερο κόστος στην επίδοση του επεξεργαστή. Αυτές είναι μηχανισμοί άμυνας βασισμένοι στο υλικό του επεξεργαστή.

Στην παρούσα διπλωματική ασχολούμαστε κυρίως με τις διάφορες κατηγορίες μηχανισμών άμυνας στο υλικό, κάνουμε μια λεπτομερή καταγραφή των διαφόρων επιθέσεων και των μηχανισμών άμυνας ενώ αξιολογούμε και συγκρίνουμε διαφόρες τεχνικές άμυνας μεταξύ τους. Οι τεχνικές άμυνας βασισμένες στο λογισμικό δεν είναι αντικείμενο έρευνας σε αυτήν τη διπλωματική και αναφέρονται επιγραμματικά.

Χωρίζουμε τους μηχανισμούς άμυνας σε τρεις βασικές κατηγορίες:

- **Τεχνικές Απόκρυψης:** Σε αυτήν την κατηγορία ανήκουν μηχανισμοί όπως το *InvisiSpec* [35], το *SafeSpec* [12] και το *Ghost Loads* [24]. Βασικό τους χαρακτηριστικό είναι ότι προσθέτουν μικρές δομές buffer δίπλα σε κάθε επίπεδο κρυφής μνήμης ούτως ώστε τα δεδομένα να πηγαίνουν εκεί μέχρι η υποθετική εκτέλεση να τελειώσει και να έχει αποσαφηνιστεί αν οι προβλέψεις ήταν σωστές ή λάθος. Στην περίπτωση που οι προβλέψεις είναι σωστές, τα δεδομένα φορτώνονται από τους buffers στην κρυφή μνήμη και η ροή εκτέλεσης εντολών συνεχίζεται κανονικά. Αν οι προβλέψεις είναι λάθος, τότε τα δεδομένα στους buffers διαγράφονται (squashed).

- **Τεχνικές Καθυστέρησης:** Σε αυτήν την κατηγορία εντάσσονται μηχανισμοί όπως το *DOLMA* [18], το *STT* [39], το *NDA* [32] και το *Delay-on-Miss* [25]. Κοινό χαρακτηριστικό τους είναι το γεγονός ότι καθυστερούν τη διάδοση μιας ¨μη ασφαλούς' εντολής μέχρι να αποσαφηνιστεί το αποτέλεσμα της υποθετικής εκτέλεσης. Μια ¨μη ασφαλή' εντολή είναι συνήθως μια εντολή φόρτωσης που έπεται από μια εντολή διακλάδωσης η οποια δεν έχει εξέλθει ακόμα από τον Re-Order Buffer (ROB).

- **Τεχνικές Αναίρεσης:** Κύριο παράδειγμα αυτής της κατηγορίας είναι ο μηχανισμός *CleanupSpec* [23]. Η κεντρική ιδέα εδώ είναι ότι η υποθετική εκτέλεση εντολών συνεχίζεται απερίσπαστα και αναιρούνται μόνο τα παράπλευρα αποτελέσματα της υποθετικής εκτέλεσης στην περίπτωση μιας λανθασμένης πρόβλεψης. Το κόστος στην επίδοση του επεξεργαστή επηρεάζεται μόνο από το μηχανισμό που εκτελεί την αναίρεση των παράπλευρων αποτελεσμάτων της υποθετικής εκτέλεσης στην περίπτωση λανθασμένης πρόβλεψης.

1.1 Θεωρητικό Υπόβαθρο

Αρχικά, δίνουμε πληροφορίες σχετικά με την δυναμική εκτέλεση εντολών εκτός σειράς και την υποθετική εκτέλεση εντολών. Στη συνέχεια εξηγούμε πώς η λανθασμένη πρόβλεψη στην υποθετική εκτέλεση εντολών μπορεί να χρησιμοποιηθεί από έναν κακόβουλο χρήστη ώστε να υπάρξει διαρροή ευαίσθητων πληροφοριών.

1.1.1 Εκτέλεση εντολών εκτός σειράς

Οι σύγχρονοι επεξεργαστές χρησιμοποιούν τη δυναμική χρονοδρομολόγηση των εντολών με το να εκτελούν ανεξάρτητες μεταξύ τους εντολές παράλληλα, εκτός της σειράς του προγράμματος, έτσι ώστε να εκμεταλλευτούν την παραλληλία σε επίπεδο εντολών για να βελτιώσουν τις επιδόσεις τους. Οι εντολές εκδίδονται σε σειρά (issued), εκτελούνται (executed) και βγάζουν τα αποτελέσματά τους εκτός σειράς προγράμματος και τελικά επικυρώνονται (retired) σε σειρά, μεταβάλλοντας και τις καταστάσεις μικροαρχιτεκτονικής.

1.1.2 Υποθετική εκτέλεση εντολών

Όταν ένα πρόγραμμα εκτελείται σε έναν μικροεπεξεργαστή, συχνά χρειάζεται να καθυστερήσει η ροή εκτέλεσής του ώστε να αποκτήσει την πληροφορία που χρειάζεται που είναι αποθηκευμένη στην κύρια μνήμη. Μια τέτοια ενέργεια όμως είναι αρκετά χρονοβόρα, κάθως το να έρθει μια εντολή από την κύρια μνήμη μπορεί να κρατήσει αρκετούς κύκλους ρολογιού με συνέπεια να υπάρξει μεγάλη επιβράδυνση στην απόδοση του επεξεργαστή. Οι πιο εξελιγμένοι επεξεργαστές των τελευταίων δεκαετιών χρησιμοποιούν την τεχνική της υποθετικής εκτέλεσης εντολών (speculative execution) ούτως ώστε να βελτιώσουν την επίδοση και την αποδοτικότητά τους. Η υποθετική εκτέλεση εντολών βελτιώνει την απόδοση του επεξεργαστή αφού εκτελούνται εντολές των οποίων η εγκυρότητα είναι αβέβαιη αντί να περιμένει το πρόγραμμα να αποσαφηνιστεί η εγκυρότητά τους. Στην περίπτωση που η πρόβλεψη της υποθετικής εκτέλεσης είναι σωστή, η εντολή φεύγει από τον Re-Order Buffer (ROB), όπου περιμένει μέχρι η υποθετική εκτέλεση να αποσαφηνιστεί, και γίνεται "retired". Στην περίπτωση που αποδειχθεί ότι η πρόβλεψη ήταν λανθασμένη, η εντολή ακυρώνεται (squashed) και η κατάσταση του επεξεργαστή γυρίζει στην προηγούμενη έγκυρη κατάσταση.

Αν θέλουμε να δώσουμε έναν σαφή αλλά συντηρητικό ορισμό του πότε μια εντολή θεωρείται ότι εκτελείται υποθετικά, θα λέγαμε ότι η εντολή αυτή θα πρέπει να εκτελεστεί πριν φτάσει στην κορυφή του ROB και εξέλθει από αυτόν. Παραδείγματος χάριν, αν μια εντολή φόρτωσης εκτελεστεί μετά από μια εντολή άλματος, η οποία δεν έχει φτάσει στην κορυφή του ROB και τελικά η αρχική πρόβλεψη ήταν λανθασμένη, τότε όταν διαπιστωθεί ότι η πρόβλεψη ήταν λανθασμένη η εντολή άλματος θα ακυρωθεί. Ωστόσο, η ακύρωση της εντολής άλματος δεν είναι αρκετή, καθώς θα πρέπει να ακυρωθούν και οι επόμενες εντολές που εν δυνάμει μπορούν να έχουν αποθηκεύσει κάποια ευαίσθητη πληροφορία στην κρυφή μνήμη. Κάτι τέτοιο δεν συμβαίνει, με αποτέλεσμα η υποθετική εκτέλεση εντολών να βελτιώνει την ταχύτητα του επεξεργαστή, αλλά να δημιουργεί και ένα μεγάλο πεδίο για να το εκμεταλλευτεί ένας κακόβουλος χρήστης και να αποσπάσει ευαίσθητες πληροφορίες του συστήματος.

1.2 Πώς λειτουργούν οι επιθέσεις;

1.2.1 Επιθέσεις Παράπλευρων καναλιών

Στις μέρες μας η ασφάλεια ενός συστήματος είναι πρωταρχικός στόχος των μηχανικών υπολογιστών. Τα σημερινά συστήματα είναι αρκετά πολύπλοκα, καθώς το υλικό, το λογισμικό και το λειτουργικό σύστημα συνεργάζονται έτσι ώστε το σύστημα να λειτουργεί αποτελεσματικά. Ωστόσο, πάντα οι κακόβουλοι χρήστες βρίσκουν και εκμεταλλεύονται τα αδύνατα σημεία ενός συστήματος. Για αυτό το λόγο, είναι μείζονος σημασίας οι μηχανικοί υπολογιστών να παρέχουν πλήρη προστασία σε κάθε βαθμίδα του συστήματος για να είναι το σύστημα ασφαλές από κακόβουλες επιθέσεις. Γενικά, υπάρχουν αρκετοί μηχανισμοί άμυνας σε επίπεδο λογισμικού που κάνουν το πρόγραμμα να τρέχει χωρίς σφάλματα ή άμυνες που αφορούν το λειτουργικό σύστημα [10]. Όμως, οι επιθέσεις παράπλευρων καναλιών τύπου είναι ικανές να παρακάμψουν όλες αυτές τις άμυνες μη αφήνοντας ίχνη [36], [21], [22], [7], [30], [13], [15], [33], [14]. Μια επίθεση παράπλευρων καναλιών μπορεί να αλιεύσει ευαίσθητες πληροφορίες από ένα σύστημα, εκμεταλλευόμενη τα παράπλευρα αποτελέσματα της εκτέλεσης του προγράμματος παρατηρόντας πώς το πρόγραμμα χρησιμοποίησε τους διαμοιραζόμενους πόρους του υλικού (π.χ. κρυφή μνήμη, προβλεπτές αλμάτων, επαναπρογραμματιζόμενες πύλες κλπ). Τα παράπλευρα αποτελέσματα που μπορεί να εκμεταλλευτεί ο επιτιθέμενος συνήθως αφορούν χρονική διάρκεια, κατανάλωση ενέργειας ή ακόμη και ήχο [6], [11], [17].

Σε αυτή τη διπλωματική επικεντρωνόμαστε στην ειδική περίπτωση των παράπλευρων καναλιών λόγω της υποθετικής εκτέλεσης εντολών, και εστιάζουμε στα παράπλευρα κανάλια κρυφής μνήμης. Από όλους τους τύπους παράπλευρων καναλιών, αυτά που αφορούν την κρυφή μνήμη είναι αυτά που προσφέρουν την πιο ευρεία επιφάνεια ούτως ώστε οι κακόβουλοι χρήστες να αναπτύξουν μια πληθώρα από επικίνδυνες επιθέσεις. Στα παράπλευρα κανάλια κρυφής μνήμης ο επιτιθέμενος έχει πρόσβαση σε ευαίσθητες πληροφορίες του συστήματος εξαιτίας της αλληλεπίδρασης της εκτέλεσης του προγράμματος με τις καταστάσεις της κρυφής μνήμης. Συγκεκριμένα, οι περισσότεροι κακόβουλοι χρήστες εκμεταλλεύονται το γεγονός ότι μια πρόσβαση στην μνήμη διαρκεί πολύ περισσότερους κύκλους αν είναι άστοχη (δηλαδή το ζητούμενο δεδομένο δε βρίσκεται στην κρυφή μνήμη) από ότι αν είναι εύστοχη. Αυτό είναι κάτι που καθιστά τις αντίστοιχες επιθέσεις παράπλευρων καναλιών πολύ επικίνδυνες, καθώς η χρονίκη διαφορά στη διάρκεια μιας ευστοχης προσβασης στην κρυφή μνήμη από μια άστοχη, είναι μια εγγενής ιδίοτητα των κρυφών μνημών.

1.2.2 Μεταβατικές Επιθέσεις Παράπλευρων Καναλιών

Στις μεταβατικές επιθέσεις παράπλευρων καναλιών οι επιτιθέμενοι εκμεταλλεύονται την υποθετική εκτέλεση εντολών, ωθώντας τους προβλεπτές αλμάτων σε λάθος πρόβλεψη δίνοντάς τους κατάλληλες εισόδους, έτσι ώστε να εξαναγκάσουν τη ροή εκτέλεσης να συνεχίσει από το λανθασμένο μονοπάτι. Η εκτέλεση από το λανθασμένο μονοπάτι της ροής εκτέλεσης, ονομάζεται μεταβατική εκτέλεση εντολών καθώς αυτή διαρκεί μέχρι τη χρονική στιγμή που αποσαφηνίζεται ότι η πρόβλεψη ήταν λανθασμένη, οι παράπλευρες ιδιότητές της αφαιρούνται λόγω της ακύρωσης των εντολών και η εκτέλεση συνεχίζει από το σωστό μονοπάτι. Όμως, κάτι τέτοιο δε συμβαίνει και γιαυτό δημιουργείται ένας τεράστιος χώρος ούτως ώστε να αναπτυχθούν διάφορες επιθέσεις.

Οι μεταβατικές επιθέσεις παράπλευρων καναλιών εξαιτίας υποθετικής εκτέλεσης εντολών ορίζονται ως οι επιθέσεις που έχουν πρόσβαση σε δεδόμένα (π.χ. μέσω μιας εντολής φόρτωσης) κατά τη διάρκεια της μεταβατικής εκτέλεσης και έπειτα μεταφέρουν την ευαίσθητη πληροφορία μέσω ενός παράπλευρου καναλιού στον επιτιθέμενο. Για να θεωρείται πετυχημένη μια μεταβατική επίθεση παράπλευρου καναλιού θα πρέπει να αποτελείται από δυο μέρη. Αρχικά, θα πρέπει, αφού η ροή εκτέλεσης του προγράμματος εισέλθει στο λανθασμένο μονοπάτι, να έχει πρόσβαση στην ευάισθητη πληροφορία (συνήθως αυτό γίνεται μέσω μιας εντολής φόρτωσης). Το δεύτερο μέρος της επίθεσης είναι εξίσου

σημαντικό, δηλαδή η μετάδοση της πληροφορίας μέσω ενός παράπλευρου καναλιού. Αν και υπάρχουν πολλά είδη παράπλευρων καναλιών που μπορεί να διαμορφωθούν, τα πιο πολυχρηστικά και πιο ευρέως εκμεταλλεύσιμα είναι τα παράπλευρα κανάλια που αφορούν την κρυφή μνήμη [36], [6], [11], [17].

1.2.3 Αιτίες Μεταβατικής Εκτέλεσης

Παρακάτω παρουσιάζουμε μερικά γεγονότα που προκαλούν την μεταβατική εκτέλεση εντολών.

- **Λανθασμένη πρόβλεψη**: Η λανθασμένη πρόβλεψη είναι η πρώτη και πιο διαδεδομένη αιτία της μεταβατικής εκτέλεσης εντολών. Οι σύγχρονοι επεξεργαστές βασίζονται στο να κάνουν πολλές προβλέψεις έτσι ώστε να επιτύχουν καλύτερη απόδοση. Εάν η πρόβλεψη είναι σωστή τότε η ροή εκτέλεσης του προγράμματος προχωράει κανονικά. Έτσι η απόδοση αυξάνεται, αφού οι εντολές εκτελούνται νωρίτερα. Εάν αποδειχθεί λανθασμένη, τότε οι εντολές κάτω από το λανθασμένο μονοπάτι θα διαγραφούν. Στο συμβάν μιας λανθασμένης πρόβλεψης είναι κυρίως βασισμένες οι επιθέσεις τύπου Spectre. Υπάρχουν τρια είδη προβλέψεων:

  (1) Προβλέψεις ροής ελέγχου προγράμματος: Αυτές προβλέπουν το μονοπάτι εκτέλεσης που θα ακολουθηθεί

  (2) Προβλέψεις διεύθυνσης: Αυτές προβλέπουν σε ποια φυσική διεύθυνση θα γίνει φόρτωση ή αποθήκευση δεδομένων.

  (3) Προβλέψεις τιμής: Για να βελτιωθεί περαιτέρω η απόδοση του επεξεργαστή, μπορεί να προβλεφθούν οι τιμές που περιμένει να λάβει μια εντολή (λόγω εξάρτησης από προηγούμενες εντολές) και να μην καθυστερεί η τεχνική διοχέτευσης.

- **Εξαιρέσεις**: Η δεύτερη πιο πιθανή αιτία μεταβατικής εκτέλεσης εντολών είναι οι εξαιρέσεις. Αν μια εντολή προκαλέσει μια εξάιρεση, η διαχείρηση της εξάιρεσης μπορεί να καθυστερήσει και έτσι να αφήσει αρκέτο χρόνο να εκτελείται μεταβατικά. Το γεγονός αυτό εκμεταλλεύονται κυρίως οι επιθέσεις τύπου Meltdown.

1.2.4 Disclosure Gadget

Ένα disclosure gadget περιλαμβάνει δυο στοιχεία:

- 1. Τη φόρτωση της ευαίσθητης πληροφορίας σε έναν καταχωρητή
- 2. Τη μεταφορά της μέσω ενός παράπλευρου καναλιού

Όπως φαίνεται και από το Σχήμα 1, για παράπλευρα κανάλια κρυφής μνήμης θα πρέπει να υπάρχει μια πρόσβαση μνήμης της οποίας η διεύθυνση εξαρτάται από την πληροφορία που θέλουμε να υποκλέψουμε.

```
struct array *arr1 = ...;
struct array *arr2 = ...;
unsigned long offset = ...;
if (offset < arr1_len) {
        sec = arr1[offset]; // 1. Load secret
        value2 = arr2[sec*c];} // 2. Encode
```

Σχήμα 1: Disclosure Gadget

1.2.5 Ανακτώντας την ευαίσθητη πληροφορία

Δυο είναι οι κύριες τεχνικές για την ανάκτηση της πληροφορίας που υποκλάπηκε από τον επιτιθέμενο. Αυτές είναι οι Prime+Probe [37] και Flush+Reload [38]. Η πρώτη τεχνική γεμίζει με τυχαία δεδομένα τα σετ της κρυφής μνήμης. Όταν το πρόγραμμα παράξει κάποιες αναφορές στη μνήμη, κάποιες προσβάσεις θα αντικαταστήσουν τα υπάρχοντα δεδομένα στην κρυφή μνήμη. Έτσι μετά ο επιτιθέμενος θα ξαναέχει πρόσβαση στην κρυφή μνήμη και θα συμπεράνει αν η εκτέλεση του προγράμματος οδήγησε στην αντικατάσταση κάποιων δεδομένων. Με τη δεύτερη ο επιτιθέμενος μπορεί να συμπεράνει ποια ήταν η ευαίσθητη πληροφορία, έμμεσα, από τη χρονική διάρκεια που κρατάει μια πρόσβαση στη μνήμη.

1.3 Πώς λειτουργεί το Meltdown

Το Meltdown, επιτρέπει στους επιτιθέμενους να έχουν πρόσβαση σε πληροφορίες ασφαλείας μέσω του λειτουργικού συστήματος ενός υπολογιστή. Αρχικά, ο επιτιθέμενος ζητάει να διαβάσει μνήμη kernel. Παρόλο που ένα τέτοιο αίτημα πρόκειται να απορριφθεί από το μηχανισμό προστασίας της μνήμης, μια λανθασμένη πρόβλεψη κατά τη διάρκεια της υποθετικής εκτέλεσης θα έχει ως αποτέλεσμα τελικά η ευαίσθητη πληροφορία να είναι ορατή στον επιτιθέμενο, αφού πρώτα την έχει μεταφέρει μέσω του παράπλευρου καναλιού προς το μέρος του.

1.4 Πώς λειτουργεί το Spectre

Το Spectre, δίνει τη δυνατότητα σε κακόβουλους χρήστες να "εξαπατήσουν" τον επεξεργαστή ώστε να ξεκινήσει η υποθετική εκτέλεση εντολών. Έτσι θα μπορούν να διαβάσουν τα ευαίσθητα δεδομένα που διαθέτει ο επεξεργαστής καθώς προσπαθεί να μαντέψει τι λειτουργία θα εκτελέσει ο υπολογιστής στη συνέχεια. Γενικά, μια επίθεση Spectre τύπου εκμεταλλεύεται το γεγονός όταν συμβεί μια λάθος πρόβλεψη. Όπως φαίνεται και στο Σχήμα 2, μια επίθεση τύπου Spectre αποτελείται από τρια μέρη. Η πρώτη φάση περιλαμβάνει ένα κακόβουλο χρήστη ή λογισμικό που δίνει επιτηδευμένα συγκεκριμένες εισόδους σε έναν προβλεπτή, έτσι ώστε να τον αναγκάσει να κάνει λάθος πρόβλεψη κατά τη διάρκεια της υποθετικής εκτέλεσης και να οδηγήσει την ροή εκτέλεσης του προγράμματος στο λάθος μονοπάτι (μεταβατική εκτέλεση). Η δεύτερη φάση αποτελείται από το disclosure gadget το οποίο πρώτα μέσω μιας εντολής φόρτωσης έχει πρόσβαση στην ευαίσθητη πληροφορία και έπειτα μέσω μιας εκ νέου εντολής πρόσβασης στην μνήμη η οποία εξαρτάται από την ευαίσθητη πληροφορία δημιουργεί το παράπλευρο κανάλι μεταφοράς της. Τέλος, η τρίτη φάση αποτελείται από την ανάκτηση της ευαίσθητης πληροφορίας μέσω του παράπλευρου καναλιού από τον επιτιθέμενο. Αυτό γίνεται συνήθως με τις τεχνικές που προαναφέραμε παραπάνω, μετρώντας κυρίως τη χρονική διάρκεια των προσβάσεων στην κρυφή μνήμη.

```
uint8 A[10];
uint8 B[256*64];
void victim (size_t addr) {
    if (addr < 10) { // mispredicted branch
M1:     uint8 val = A[addr]; // secret is accessed
M2:     ... = B[64 * val]; // secret is transmitted
    }
}
```

Σχήμα 2: Spectre επίθεση τύπου 1

1.5 Μηχανισμοί Άμυνας βασισμένοι στη Σχεδίαση Υλικού

Η πιο απλή και συναμά εξαιρετικά αποτελεσματική λύση σχετικά με την ανάσχεση επιθέσεων παράπλευρων καναλιών εξαιτίας της υποθετικής εκτέλεσης εντολών είναι η εκτέλεση σε σειρά. Όμως, παρόλο που σε αυτή την περίπτωση θα ανακόπταμε καθολικά όλες τις επιθέσεις που εκμεταλλεύονται τα παράπλευρα αποτελέσματα της υποθετικής εκτέλεσης εντολών, το τίμημα σε κόστος επίδοσης θα ήταν μεγάλο. Έτσι, οι επεξεργαστές θα γίνονταν εξαιρετικά αργοί και άρα μη πρακτικοί. Κάποιες υπάρχουσες λύσεις βασισμένες στο λογισμικό κρίνονται είτε ανεπαρκείς, καθώς προστατεύουν μόνο κάποιες πολύ ειδικές περιπτώσεις αυτών των επιθέσεων, είτε έρχονται με ένα μεγάλο τίμημα σε κόστος επίδοσης [10]. Για αυτό το λόγο, οι μηχανικοί υπολογιστών στράφηκαν στην αναζήτηση πιο ριζοσπαστικών λύσεων, προσπαθώντας να επέμβουν δραστικά στον τρόπο με τον οποίο είναι δομημένος ο σύγχρονος επεξεργαστής και κοιτώντας προς στη μεριά του υλικού.

Έως τώρα υπάρχουν τριών ειδών τεχνικών άμυνας που βασίζονται στο υλικό: τεχνικές απόκρυψης, τεχνικές καθυστέρησης και τεχνικές αναίρεσης.

1.5.1 Τεχνικές Απόκρυψης

Σε αυτήν την κατηγορία ανήκουν μηχανισμοί άμυνας όπως το InvisiSpec [35], το SafeSpec [12] και το Ghost Loads [24]. Το κύριο χαρακτηριστικό τους είναι ότι φορτώνουν τα δεδομένα σε ειδικά σχεδιασμένους καταχωρητές, που βρίσκονται δίπλα από την κρυφή μνήμη, μέχρις ότου η υποθετική εκτέλεση εντο-

λών αποσαφηνιστεί. Στην περίπτωση που η αρχική πρόβλεψη ήταν σωστή, τότε τα δεδομένα (ξανά)φορτώνονται από τους ειδικά σχεδιασμένους καταχωρητές στην κρυφή μνήμη. Εάν η αρχική πρόβλεψη ήταν λανθασμένη τότε τα δεδομένα διαγράφονται από τους ειδικά σχεδιασμένους καταχωρητές και το πρόγραμμα συνεχίζει από το σημείο πριν γίνει η λανθασμένη πρόβλεψη. Θα ασχοληθούμε εκτενέστερα με τον μηχανισμό Invisispec , καθώς θεωρείται ο πρώτος μηχανισμός που παρέχει καθολική άμυνα από επιθέσεις παράπλευρων καναλιών κρυφής μνήμης λόγω της υποθετικής εκτέλεσης εντολών. Στο αγγλικό κείμενο μπορεί να βρει κανείς λεπτομέρειες και για άλλους μηχανισμούς διαφορετικών κατηγοριών.

Ανασκόπηση InvisiSpec

(1) Μη ασφαλείς εντολές φόρτωσης κατά τη διάρκεια υποθετικής εκτέλεσης: Μια εντολή φόρτωσης πριν φτάσει την κορυφή του Re-Order Buffer (ROB) θεωρείται ως υποθετική εντολή εκφόρτωσης. Αυτές οι εντολές στο *InvisiSpec* ορίζονται ως μη ασφαλείς υποθετικές εντολές εκφόρτωσης (Unsafe Speculative Loads - USLs) και ο αυστηρός ορισμός τους εξαρτάται από το μοντέλο επίθεσης. Το InvisiSpec παρουσιάζει δυο μοντέλα επίθεσης: Το Spectre και το Futuristic. Στο πρώτο, ως USLs θεωρούνται οι εντολές φόρτωσης που ακολουθούν μια εντολή άλματος η οποία ακόμη δεν έχει εξέλθει από τον ROB. Στο Futuristic, ως USLs θεωρείται κάθε υποθετική εντολή φόρτωσης εκτός από αυτές που είναι αδύνατο να ακυρωθούν από μια προηγούμενη εντολή. Οι εντολές φόρτωσης γίνονται ¨ασφαλείς¨, όταν στην πρώτη περίπτωση οι εντολές άλματος αποσαφηνιστούν ή όταν στη δεύτερη, οι εντολές φόρτωσης δε είναι πλέον υποθετικές.

(2) Κάνει τα USLs αόρατα: Όπως αναφέρθηκε και πριν, η βασική ιδέα αυτού του μηχανισμού είναι να κάνει αόρατα τα USLs στην ιεραρχεία της κρυφής μνήμης. Αυτό το πετυχαίνει με το να αποθηκεύει τις μη ασφαλείς εντολές φόρτωσης στους ειδικά σχεδιασμένους καταχωρητές, έτσι ώστε να μη φανούν και οι παράπλευρες ιδιότητες της υποθετικής εκτέλεσης στον επιτιθέμενο. Όταν η εντολή θεωρηθέι ασφαλής, τότε και μόνο τότε, θα φορτωθεί εκ νέου από τους ειδικά σχεδιασμένους καταχωρητές στην μνήμη. Το σημείο στο οποίο μια εντολή φόρτωσης γίνεται ασφαλής και έχει ελεγχθεί η εγκυρότητά της ως προς το αντίστοιχο μοντέλο συνέπειας μνήμης, ονομάζεται "σημείο ορατότητας" (visibility point). Το σημείο ορατότητας εξαρτάται από το μοντέλο επίθεσης που θεωρούμε κάθε φορά.

(3) Διατηρώντας τη συνέπεια της μνήμης: Είναι σημαντικό να

διατηρείται η συνέπεια της μνήμης αφού η εντολή φόρτωσης γίνει ασφαλής. Για αυτό το λόγο, το *InvisiSpec* πριν ξαναφορτώσει τα δεδομένα από τους ειδικά σχεδιασμένους καταχωρητές στην μνήμη, κάνει πρώτα έναν έλεγχο εγκυρότητας, ώστε να ικανοποιείται το εκάστοτε μοντέλο συνέπειας μνήμης.

1.5.2 Τεχνικές Καθυστέρησης

Εδώ ανήκουν μηχανισμοί άμυνας όπως το *DOLMA* [18], το *Speculative Taint Tracking (STT) [39]*, το *NDA* [32] και το *Delay-on-Miss* [25]. Το κύριο χαρακτηριστικό αυτής της κατηγορίας είναι ότι κάποιες εντολές κάτα τη διάρκεια υποθετικής εκτέλεσης θεωρούνται μη ασφαλείς, κάτω υπό συγκεκριμένες συνθήκες, και καθυστερούνται μέχρι να αποσαφηνιστεί η υποθετική εκτέλεση. Γενικά, προλαμβάνουν τη διάδοση των ¨μη ασφαλών' εντολών βασιζόμενοι στην παρατήρηση ότι μια επιτυχημένη επίθεση παράπλευρων καναλιών εξαιτίας υποθετικής εκτέλεσης πρέπει να αποτελέιται από δυο μέρη: (1) Από μια παράνομη εντολή φόρτωσης που θα έχει πρόσβαση στην ευαίσθητη πληροφορία που θα υποκλαπεί και (2) μια ή περισσότερες εντολές που είναι εξαρτώμενες από την προαναφερθείσα παράνομη εντολή φόρτωσης και διαμορφώνει το παράπλευρο κανάλι που μεταφέρει την ευαίσθητη πληροφορία στον επιτιθέμενο. Μόλις οι εντολές θεωρηθούν ξανά ως ασφαλείς, τότε οι μηχανισμοί αφήνουν την ροή εκτέλεσης του προγράμματος να συνεχίσει. Έτσι, αποφεύγεται η διάδοσή τους και η υποκλοπή τους από κάποιον κακόβουλο χρήστη.

Το *STT* για παράδειγμα, ¨στιγματίζει' εντολές πρόσβασης στην μνήμη (π.χ. εντολές φόρτωσης που έχουν πρόσβαση σε κάποια ευαίσθητη πληροφορία) και τις ¨καθαρίζει' μόλις αυτές θεωρηθούν ασφαλείς (όταν π.χ όλοι οι εξαρτώμενοι τελεστέοι της εντολής γίνουν και αυτοί ¨καθαροί¨). Με αυτόν τον τρόπο, παρόλο που επιτρέπεται η συνέχιση της εκτέλεσης και της διάδοσης μιας εντολής φόρτωσης που έχει πρόσβαση στην ευαίσθητη πληροφορία, η εκτέλεση των εξαρτώμενων εντολών από αυτήν αποτρέπεται (μέσω της καθυστέρησης) και άρα προλαμβάνεται η διαμόρφωση του παράπλευρου καναλιού που θα μεταδόσει την ευαίσθητη πληροφορία στον επιτιθέμενο. Αρκετά παρόμοιο σχεδιασμό με το *STT* έχει και το *DOLMA* , προσφέροντας επιπλέον προστασία αφού προστατεύει αποτελεσματικά και από περιπτώσεις που έχουμε Store-to-Load forwarding, κάτι που δεν προσφέρει το *STT*.

1.5.3 Τεχνικές Αναίρεσης

Σε αυτήν την κατηγορία ανήκει ο μηχανισμός *CleanupSpec* [23]. Εδώ ακουλουθείται μια εντελώς διαφόρετική προσέγγιση σε σχέση με τις τεχνικές απόκρυψης. Εδώ δεν υπάρχουν ειδικά σχεδιασμένοι καταχωρητές να αποθηκεύονται τα δεδομένα μέχρις ότου η υποθετική εκτέλεση αποσαφηνιστεί. Αντίθετα, η υποθετική εκτέλεση εντολών συνεχίζει απερίσπαστα την ροή της και μόνο στην περίπτωση που υπάρχει μια λάθος πρόβλεψη, τότε ο μηχανισμός αναιρεί όλα τα παράπλευρα αποτελέσματα της υποθετικής εκτέλεσης και συνεχίζεται η ροή του προγράμματος από το προηγούμενο σωστό σημείο πρόβλεψης.

1.6 Μεθοδολογία

Για την εξαγωγή των αποτελεσμάτων χρησιμοποιήσαμε εκτενώς τον προσομοιωτή gem5. Το gem5 είναι ένας προσομοιωτής τελευταίας τεχνολογίας που δίνει τη δυνατότητα για ακριβείς προσομοιώσεις hardware εξαρτημάτων του υπολογιστή ανά κύκλο ρολογιού. Χρησιμοποιείται ευρέως στον ακαδημαϊκό χώρο αλλά και στη βιομηχανία για την έρευνα και ανάπτυξη νέων τεχνολογιών του υλικού. Είναι προγραμματισμένο στις γλώσσες C++ και Python. Υποστηρίζει εξαιρετικά πολλές αρχιτεκτονικές με κυριότερες τις x86 και ARM. Επίσης, ικανοποιεί τις απαιτήσεις μας για την προσομοίωση των επιθέσεων, των αρχιτεκτονικών x86 και ARM, της κρυφής μνήμης, της εκτέλεσης εκτός σειράς, των προβλεπτών αλμάτων κλπ.

Υπάρχουν δυο είδη προσομοιώσεων που μπορεί να τρέξει ένας χρήστης στον gem5 προσομοιωτή: σε System-call Emulation (SE) και σε Full-System (FS). Το συγκριτικό πλεονέκτημα του πρώτου είδους έναντι του δεύτερου είναι η κατά πολύ μικρότερη χρονική διάρκεια που απαιτεί η προσομοίωση, καθώς δεν απαιτείται η προσομοίωση του λειτουργικού συστήματος. Γενικά, μια προσομοίωση μπορεί να κρατήσει μέρες, εβδομάδες ή σε ακραίες περιπτώσεις και μήνες. Επομένως, για να μπορέσουμε να τρέξουμε διάφορα πειράματα αρκετές φορές σε εύλογο χρονικό διάστημα θα προτιμήσουμε την SE mode προσομοίωση. Ωστόσο, τα αποτελέσματα που θα πάρουμε δεν θα είναι το ίδιο ακριβή σε σχέση με την Full System προσομοίωση η οποία προσομοιώνει και το λειτουργικό σύστημα παρέχοντας μεγαλύτερη ακρίβεια, άλλα είναι σημαντικά πιο αργή.

Στα πειράματά μας, χρησιμοποιήσαμε τον προσομοιωτή gem5 με τις παρακάτω παραμέτρους όπως φαίνονται στον Πίνακα 1. Επίσης χρησιμοποιήσαμε

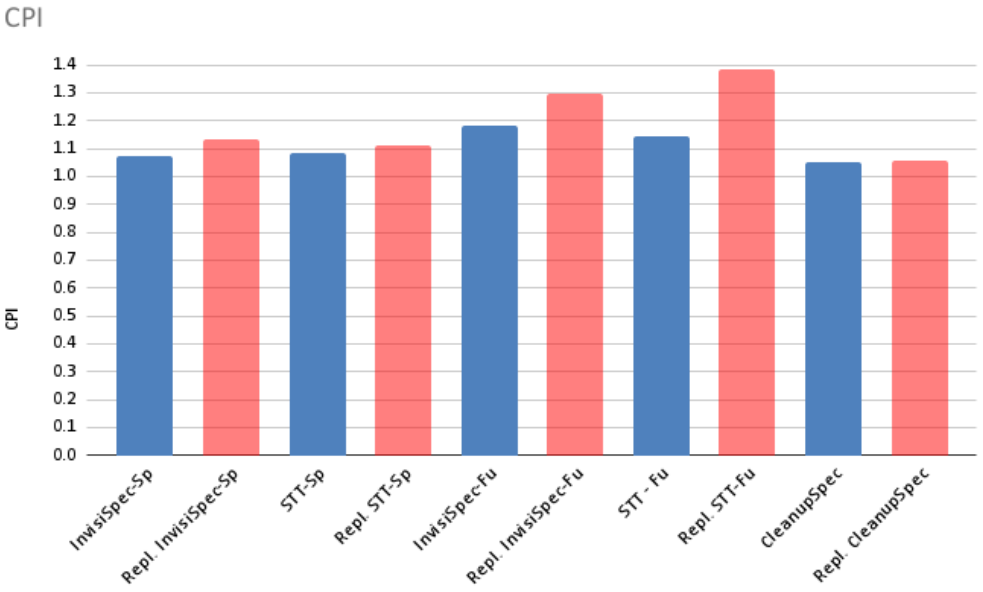
Parameter	**Value**
Architecture	1 core (SPEC) at 2.0GHz
Core (OoO)	8-issue, out-of-order, no SMT, 32 Load Queue entries, 32 Store Queue entries, 192 ROB entries, Tournament branch predictor, 4096 BTB entries, 16 RAS entries
Core (in-order)	TimingSimpleCPU from gem5
Private L1-I Cache	32KB, 64B line, 4-way, 1 cycle round-trip (RT) lat., 1 port
Private L1-D Cache	64KB, 64B line, 8-way, 1 cycle RT latency, 3 Rd/Wr ports
Shared L2 Cache	Per core: 2MB bank, 64B line, 16-way, 8 cycles RT local latency, 16 cycles RT remote latency (max)
Network	4×2 mesh, 128b link width, 1 cycle latency per hop
Coherence Protocol	Directory-based MESI protocol
DRAM	RT latency: 50 ns after L2

Πίνακας 1: Παράμετροι της προσομοιωμένης αρχιτεκτονικής

την σουίτα των μετροπρογραμμάτων SPEC CPU2006 benchmarks για την αξιολόγηση της επίδοσης. Συγκεκριμένα, για τα μετροπρογραμματα χρησιμοποιήσαμε τα reference input από τις εισόδους και εκτελέσαμε προσομοίωση σε SE mode, κάνοντας παράλειψη τις πρώτες 10 δισεκατομμύρια εντολές και μετά ακολουθήσαμε εκτός σειράς εκτέλεση για το επόμενο 1 δισεκατομμύριο εντολές.

1.7 Πειραματική Αξιολόγηση Επίδοσης

Εστιάζουμε σε ένα μηχανισμό από κάθε κατηγορία. Συγκεκριμένα, εστιάζουμε στο μηχανισμό *InvisiSpec* από την κατηγορία των τεχνικών απόκρυψης, το *DOLMA* από την κατηγορία των τεχνικών καθυστέρησης και το *CleanupSpec* από τις τεχνικές αναίρεσης. Στο ελληνικό κείμενο επιλέγουμε να αξιολογήσουμε μόνο τα αποτελέσματα του μηχανισμού *InvisiSpec*. Περισσότερες πληροφορίες για την αξιολόγηση των αποτελεσμάτων των άλλων δυο μηχανισμών μπορο-



Σχήμα 3: Με μπλε τα original αποτελέσματα και με κόκκινο τα δικά μας αποτελέσματα

ύν να βρεθούν στο αγγλικό κείμενο. Στο Σχήμα 3 βλέπουμε τα αποτελέσματα των αρχικών αποτελεσμάτων κάθε μηχανισμού που δίνουν οι συυγραφείς του με τις μπλε στήλες και με κόκκινο βλέπουμε τα αποτελέσματα που εξάγαμε εμείς. Οι πρώτες τέσσερεις στήλες αντιπροσωπεύουν το overhead που προκαλείται για προστασία μόνο από Spectre τύπου επιθέσεις, ενώ οι υπόλοιπες έξι αφορούν το overhead που εισάγουν στον επεξεργαστή οι μηχανισμοι για προστασία από επιθέσεις Spectre και Meltdown τύπου.

Αξιολόγηση Επίδοσης Μηχανισμού InvisiSpec

Συγκρίνουμε τον χρόνο εκτέλεσης των 21 μετροπρογραμμάτων της σουίτας SPEC CPU2006 για 5 διαφορετικά configurations. *Base*, είναι ο απλός, μη ασφαλής επεξεργαστής χωρίς καμία μετατροπή. Οι παραλλαγές *Fe-Sp* και *Fe-Fu* τοποθετούν απλά ένα fence πριν από κάθε εντολή άλματος ή πριν από κάθε εντολή φόρτωσης, αντίστοιχα. Τα configurations *IS-Sp* και *IS-Fu* είναι οι δυο παραλλαγές προστασίας που προσφέρει το *InvisiSpec* για το Spectre attack model και το Futuristic attack model αντίστοιχα. Όλες οι στήλες κάθε

configuration είναι ομαλοποιημένες στο Base.

Αν εστιάσουμε στις fence-based τεχνικές άμυνας διαπιστώνουμε ότι έχουν τεράστιο κόστος στην επίδοση της τάξεως των 100% και 250% για τους σχηματισμούς Fe-Sp και Fe-Fu αντίστοιχα. Σε σύγκριση με τις fence-based λύσεις, το *InvisiSpec* προσφέρει σαφώς βελτιωμένες επιδόσεις. Ο μέσος χρόνος εκτέλεσης για τα 21 μετροπρογράμματα που χρησιμοποιήθηκαν είναι 7.6% και 18.2% για το IS-Sp και το IS-Fu configuration, αντίστοιχα.

Αξιολογώντας την επίδοση του μηχανισμού *InvisiSpec*, παρατηρούμε ότι το μετροπρόγραμμα που παρουσιάζει τη μεγαλύτερη επιβράδυνση στην επίδοση σε σχέση με τον απλό επεξεργαστή είναι το *omnetpp*. Το *omnetpp* παρουσιάζει επιβράδυνση της τάξης του 185% και 200% για το IS-Sp και το IS-Fu, αντίστοιχα. Πιθανότατα, ο κύριος λόγος που συμβαίνει αυτό είναι ότι το *omnetpp* είναι ένα μετροπρόγραμμα με πολύ μεγάλο TLB Miss Rate. Ως γνωστόν, οι αστοχίες στο TLB είναι αρκετά κοστοβόρες σε κύκλους ρολογιού και έτσι επηρεάζουν αρκέτα την συνολική απόδοση του συγκεκριμένου μετροπρογράμματος. Συγκεκριμένα, στο Base configuration, τα TLB misses δεν καθυστερούνται καθόλου, ενώ στα configurations που αφορούν τις δυο παραλλαγές του μηχανισμού InvisiSpec, καθυστερούνται μέχρις ώτου φτάσουν στο visibility point. Επίσης, το *omnetpp* έχει μεγάλο ποσοστό αστοχίας πρόβλεψης στις εντολές άλματος αλλά και μεγάλο ποσοστό αστοχίας στην Last Level Cache, γεγονότα που χειροτερεύουν ακόμη περισσότερο την επίδοση του *omnetpp*.

Ένα μετροπρόγραμμα που παρουσιάζει εξίσου μεγάλο slowdown στην επίδοσή του είναι το *libquantum*. Δυο κατά τη γνώμη μας είναι οι βασικότεροι λόγοι που συμβαίνει αυτό. Αρχίκα το *libquantum* έχει πολύ μεγάλο ποσοστό αστοχίας στην L1 D-Cache, σχετικά με τα υπόλοιπα μετροπρογράμματα της σουίτας. Ο δεύτερος λόγος είναι ότι η επίδοση του *libquantum* βελτιώνεται σημαντικά όταν υποστηρίζεται και η λειτουργία της προανάκτησης εντολών (prefetching), κάτι που στο μηχανισμό του InvisiSpec δεν ισχύει.

Μελλοντικές Κατευθύνσεις

Ασχοληθήκαμε εκτενώς και με το TLB κομμάτι. Διαπιστώσαμε ότι ο προσομοιωτής gem5 στην SE mode προσομοίωση δεν μοντελοποιεί με ακρίβεια τις καθυστερήσεις όταν έχουμε μια εύστοχη ή άστοχη πρόσβαση στο TLB. Για να έχουμε μεγαλύτερη ακρίβεια αποτελεσμάτων θα έπρεπε να εκτελούσαμε Full System προσομοιώσεις, όμως αυτό θα έκανε πολύ πιο χρονοβόρες τις προσομοιώσεις μας. Για αυτό το λόγο, προσπαθήσαμε να μοντελοποιήσουμε σωστά

και ρεαλιστικά τις καθυστερήσεις αυτές, κυρίως την καθυστέρηση της αστοχίας στο TLB. Αυτό θα μπορεί να μας οδηγήσει στο μέλλον σε ακριβέστερα αποτέλεσματα όταν αξιολογούμε τα μετροπρογράμματά μας.

2 Introduction

Computer architects have been working consistently hard throughout the decades to improve the performance and the efficiency of processors. Speculative execution, one of the most fundamental techniques that has been introduced to achieve higher performance in modern processors, has been shown to have significant security implications to be considered. The reason is that speculative execution of instructions causes the micro-architectural state of the processor to be modified. Afterwards, an adversary exploits the side effects of speculative execution and the footprint left. Until recently, this was not thought to have security implications, as incorrect speculation is guaranteed to be squashed by the hardware. However, the recent disclosure of Spectre and Meltdown in January 2018 [13],[15], as well as the discovery of other attacks/sources of vulnerabilities [30], [36], [1], [5], [7], [9], [21], [28], [14], [22], [26], [33] has uncovered severe vulnerabilities in processors that systems are still struggling to deal with.

2.1 Motivation

Many software mitigation solutions have been proposed to address that problem. However, those software-based solutions that currently exist tend to have incomplete coverage [10], [4]. The main reason that software mitigation techniques fail to provide complete protection is that Spectre and Meltdown type attacks exploit some fundamental concepts that the processor is relied on and they are completely independent from the OS. While software-based defenses are considered out of scope in this thesis, we briefly discuss them and outline their limitations that have motivated research community to explore hardware support (Section 5.4).

As a logical consequence, these attacks have raised a lot of interest, and motivated computer architects to rethink the way they design modern processors and propose a number of hardware mitigation mechanisms. Indeed, a plethora of hardware-based mitigation mechanisms has been recently proposed. Although they provide a software-transparent approach to address the problem of speculative execution attacks, they may introduce excessive overhead affecting the performance of the running applications [39], [32]. Therefore, it is highly essential for computer architects to propose new defenses–

mainly at micro-architecture level–that incur low performance overhead, as well as they maintain the security of the processor.

2.2 Approach

In this diploma thesis we focus on the hardware-based mitigation mechanisms, we classify them based on their mitigation approach, and we evaluate quantitatively a mechanism from each category. More specifically, we categorize the proposed hardware mitigation mechanisms into three classes:

- **Hiding**-based solutions. Namely, *InvisiSpec* [35], *SafeSpec* [12] and *Ghost Loads* [24] mechanisms are in this category. The common theme in all these defenses is that they store the speculative data in specially designed buffers until the speculation is resolved.
- **Delaying**-based defenses. In particular, *DOLMA* [18], *STT* [39], *NDA* [32] and *Delay-on-Miss* [25] are mitigation schemes that are based on the concept of restricting the forwarding of unsafe data.
- **Undoing**-based solution. Characteristically, *CleanupSpec* [23] mechanism follows this approach. The main idea is the permission of speculative execution to proceed unhindered and undo any side-effects in an event of mis-speculation.

Based on the aforementioned classification, we pick one proposal from each class and describe/analyze it in more detail. We focus on *InvisiSpec* from the first category, *STT* and *DOLMA* from the second category, and *CleanupSec* from the third one.

For our evaluation, we replicated the statistics results of one defense mechanism from each category by using the gem5 Simulator running in SE mode the SPEC CPU2006 benchmarks. Following the same configuration setup of the simulator as the authors of the original publications used, our results are relatively close with those of the original papers.

In summary, the main contributions of this diploma thesis are:

- We summarize and categorize the proposed hardware mitigation mechanisms for protecting against side-channel attacks due to speculative execution.

- We quantitatively evaluate and compare the performance impact of three mitigation mechanisms that belong to the three different identified classes.

Organization of the document

In Section 3 we provide a general background of Out-of-Order and speculative execution. In Section 4 we explain how the attackers can cause sensitive information leakage and we analyze how Spectre and Meltdown work. In Section 5 we summarize and categorize proposed mitigation mechanisms into three main classes: hiding-based, delaying-based and undo-based techniques. In Section 6, we present the methodology we followed and in Section 7 we discuss in detail our evaluation results of mitigation schemes from each category. Finally, this diploma thesis makes conclusions in Section 8 and discusses possible future directions in Section 9.

3 Background

We first provide background on speculative execution in modern out-of-order processors. We then describe how transient (i.e., mis-speculated) execution can be exploited to potentially leak secrets.

3.1 Out-of-Order Execution

Dynamically-scheduled processors execute data-independent instructions in parallel, out of program order, and thereby exploit instruction-level parallelism in order to improve performance. While fetching and decoding of instructions is typically performed in order of the (speculative) program stream, their execution may be allowed to occur out-of-order, before then being retired in true program order.

3.2 Speculative Execution

When a program executes on a microprocessor, it has to often wait to get the information from main memory. However, compared to execution time on the microprocessor, the fetch time from memory is long. Modern processors leverage the advantages of speculative execution in order to further improve their efficiency and their speed. Speculative execution improves performance by executing instructions whose validity is uncertain instead of waiting to determine their validity. If such a speculative instruction turns out to be valid, it is eventually retired; otherwise, it is squashed and the processor's state is rolled back to a valid state.

In modern out-of-order cores, if the direction or target of a branch is mispredicted, then this misprediction may cause a large number of future, incorrect instructions to be executed before being thrown away.

Strictly speaking, an instruction executes speculatively in an out-of-order processor if it executes before it reaches the head of the ROB. There are multiple reasons why a speculatively-executed instruction may end up being squashed. For instance, one reason is that a preceding branch resolves and turns out to be mispredicted. This is critical when hardware state can be impacted by the loading of secret data. If this secret data can be used as

input to other instructions, it can be indirectly leaked even if only accessed speculatively. Thus, speculative execution while improves the speed of the processor, it also renders it vulnerable to various side-channel attacks.

4 How The Attackers Work?

Speculative execution attacks exploit the side effects of transient instructions, i.e., speculatively-executed instructions that are destined to be squashed.

4.1 Side Channel Attacks

Nowadays, security has become a key consideration in modern systems. Modern systems can be very complicated as software, hardware and host operating system work synergistically in order the system run efficiently. Attackers are smart and always try to exploit the weakest point of the system. Therefore, it is critical to provide full system protection and security at every layer of the system. There are many defenses at the software level, that make the program run bug-free or other defenses which the Operating System provides isolation between different processes and Virtual Machines. However, side channel attacks can bypass all software security policies and leave no traces. Generally speaking, in computer security, a side channel attack leaks information from the side effects of a victim program's execution on a computer system by observing how the victim used shared hardware resources (i.e. Floating Point Unit, cache, Network of Chip, branch predictors, FPGAs). These side effects include timing information, micro-architecture states, power consumption, electromagnetic leaks, or even sound. Side channels exist more generally [6], [11], [17] than the speculative attacks we focus on.

This thesis is concerned with the important type of side channel attack that exploits the shared cache hierarchies. This attack is called *cache-based side channel attack*. Among all the side channels, caches offer one of the most broad and problematic attack surfaces. In a cache-based side channel attack, an attacker obtains secret information from a victim based on the interaction between victim's execution and cache states. More specifically, most attacks exploit the difference in the access times of cache hits and misses. It is extremely challenging to eliminate cache side channels efficiently, mainly because caches are essential to processor performance, and timing difference is an inherent property of cache structures.

4.2 Transient Execution Attacks

The disclosure of recent transient execution attacks have called all software defense mechanisms into question. In transient execution attacks, attackers exploit hardware speculation (e.g. by mistraining the branch predictor) to cause the execution of instructions in incorrect paths. The execution of the instructions down the incorrect speculated path is called *transient execution* because the instructions execute transiently and should ideally disappear with no side-effects in case of a mispeculation event. When a mispeculation is detected, the architectural and micro-architectural side effects should be cleaned up but it is not done so today, leading to a number of publicized transient execution attacks that leak data across different security boundaries in computing systems. [7], [36], [21], [28], [5], [1], [30], [22], [14].

Transient execution attacks are defined as attacks that access data during transient execution and then leverage a covert channel to leak information. A transient execution attack in order to be successful consists of two parts: First, a way of bypassing software/hardware barriers to access information illegally, and second a way of leaking that secret data across the speculation boundary. The first part depends on the aforementioned property of speculative execution that allows instructions to execute and access data that they would normally not be allowed to. Generally, based on the first component, such attacks can be split into two broad categories, Spectre-style and Meltdown-style attacks, depending on how they exploit speculative execution to illegally access information. However, to leak the data, the second part is equally necessary as the first one, which takes advantage of micro-architectural state-changes that are done under speculative execution. It can be observed by the attacker during or after the speculation has been resolved and finally form the covert side channel to leak the information to the attacker. Specifically, for the second part, a number of different side-channels are available, capable of leaking information across software and hardware barriers These might include side-channels such as memory-timing side channels, port contention side channels, DRAM, etc [1], [20]. Due to being easy to exploit and also offering great versatility, memory-timing side channels (including cache-based timing side channels) are particularly popular [36], [11], [6], [17].

Therefore, from the software development perspective, software defense solutions become ineffective because speculative execution can cause exe-

cution to proceed in ways that were not intended by the programmer or compiler.

4.2.1 Causes of Transient Execution

The following is an exhaustive list of possible causes of transient execution (i.e., causes of pipeline squashing) [34].

Mis-Prediction: Mis-prediction it is the first possible and most common cause of transient execution. Modern processors are based heavily on making various predictions during the execution of a workload in order to achieve better performance and efficiency by making full use of the pipeline. If the prediction is correct, the program continues its execution and the predicted results will be used. In this way, predictions boost performance by executing instructions earlier. In the case of a mis-prediction, the code executed down the incorrect path will be squashed. There are three types of predictions: control flow predictions, address speculation and value prediction.

- **Control Flow Prediction:** Control flow predictions predict the execution path that a program will follow. Branch prediction unit (BPU) stores the history of past branch directions and targets and predicts whether the upcoming branch is taken or not by using the pattern history table (PHT) and which is the target address by using branch target buffer (BTB) or return stack buffer (RSB).

- **Address Speculation:** Address speculation is a prediction when the physical address is not fully available yet and it is used to improve the performance of the memory system. For instance, many modern Intel processors use the Store-to-Load forwarding technique in order to boost their performance.

- **Value Prediction:** In order to further improve performance, while the pipeline is waiting for the data to be loaded from memory hierarchy on a cache miss, value prediction units have been designed to predict the data value and to continue the execution based on the prediction. While this is not known to be implemented in commercial architectures, speculative execution based on value prediction had been proposed in the literature by Gabbay et al. [8]

Exceptions: The second possible and most common cause of transient execution to occur are exemptions. If an instruction causes an exception, the handling of the exception is sometimes delayed until the instructing is retired, allowing code to (transiently) execute until the exception is handled.

4.2.2 Covert Channels

A covert channel is always required in order for the attacker to obtain the secret information in architectural states. There are two parties involved in a covert channel: the sender and the receiver. In the covert channels, the sender execution will change some micro-architectural state and the receiver will observe the change to extract information, e.g., by observing the execution time.

Any sharing of hardware resources between users could lead to a covert channel between a sender and a receiver. The receiver can observe the status of the hardware with some metadata from the covert channel, such as the execution time, values of registers, system behavior, etc. The most commonly used observation by the receiver of the covert channels is the timing of execution. To observe the hardware states via timing, a timer is needed. In x86, *rdtscp* instruction can be used to read a high-resolution time stamp counter of the CPU, and thus, can be used to measure the latency of a chosen piece of code.

4.2.3 Disclosure Gadget

The covert channel is used in the disclosure gadget to transfer the secret to be accessible to the attacker architecturally. Disclosure gadget usually contains two steps:

1. load the secret into a register;
2. encode the secret into a covert channel.

As shown in Figure 1, the disclosure gadget code depends on the covert channel used. For covert channels in the memory hierarchy (e.g., cache side channel), it will consist of memory access whose address depends on the secret value.

The time resolution of the receiver is a critical metric to take seriously

```
struct array *arr1 = ...;
struct array *arr2 = ...;
unsigned long offset = ...;
if (offset < arr1_len) {
        sec = arr1[offset]; // 1. Load secret
        value2 = arr2[sec*c];} // 2. Encode
```

Figure 4: Disclosure Gadget

Covert Channel Type	Required Time Resolution of the Receiver (CPU cycles)
L1	5 vs 15
LLC	500 vs 800
TLB	105 vs 130
PHT	65 vs 90
BTB	56 vs 65
STL	30 vs 300
DRAM	300 vs 350

Table 2: Known covert channels and their required time resolution. [34]

into consideration when we compare covert channels. For a timing channel, the time resolution of the receiver's clock determines whether the receiver can observe the difference between the sender sending 0 or 1, or when there was a cache hit or cache miss. Some channels require a much higher resolution clock to measure and differentiate cycles than some others. As shown in Table 1, the covert channel in L1 cache requires a very high resolution clock to differentiate 5 cycles from 15 cycles, while the LLC covert channel onyl needs to differentiate 500 cycles to 800 cycles and thus the receiver only needs a coarse-grained clock.

4.2.4 Recovering the secret

There are two main timing-based techniques for the attacker to recover the secret that are broadly used. These are *Prime+Probe* and *Flush+Reload*.

These two techniques in most of the cases need to have a shared address space and target the Last Level Cache (LLC). *Prime + Probe* measures the time needed to read data from memory pages associated with individual cache sets. *Flush+Reload* is special variant of the more generic *Prime+Probe* because it detects the accesses to specified cache lines, while the latter technique identifies accesses to larger classes of memory locations such as cache sets. Consequently, Flush+Reload has higher accuracy, does not suffer from false positives and does not require additional processing for detecting accesses. The second advantage of Flush+Reload technique is that it focuses on the Last Level Cache, which is the cache level furthest from the processor cores (e.g L3 in processors with three cache levels). As it is shown in Table 1, Last Level Cache requires a lower resolution timer than the L1 requires and thus it is a covert channel more feasible to be successfully exploited by the attackers.

The Prime+Probe Technique

Prime and Probe [37] is one of the most general attack strategies because it does not require shared memory pages with the victim. At high level, the attacker starts with completely filling (*prime*) the cache sets he/she wish with arbitrary data to monitor using a carefully chosen eviction set. When the victim generates memory references, its accesses replace some of the cache lines in the eviction set filled by the attacker. The attacker can then access the eviction set again (*probe*); whenever an access results in a cache miss, she can infer that the victim has accessed that cache set resulting in her data being replaced.

The Flush+Reload Technique

A round of Flush+Reload [38] attack consists of three phases. During the first phase, the monitored memory line is flushed from the cache hierarchy. The spy, then, waits to allow the victim time to access the memory line before the third phase. In the third phase, the spy reloads the memory line, measuring the time to load it. If during the wait phase the victim accesses the memory line, the line will be available in the cache and the reload operation will take a short time (because of a cache hit). If, on the other hand, the victim has not accessed the memory line, the line will need to be brought from memory and the reload will take significantly longer.

4.3 How Meltdown works

Meltdown exploits a race condition between memory access and privilege level checking while an instruction is being processed. An attempt to access unauthorized memory will cause an exception and privilege level checks can be bypassed, allowing access to memory used by an operating system, or other running processes. For example, assume a user application that tries to read kernel memory. Although such request will be eventually denied, the speculatively executed instructions will result in loading of requested data into caches. Using a side channel, the attacker can effectively read arbitrary kernel (or hypervisor) memory. This is a very powerful attack, since typically kernel memory contains a direct mapped region allowing the attacker to dump the entire physical memory on a given system. Since the exception eventually will be raised, this attack requires the ability to tolerate and recover from segmentation faults.

Meltdown attacks are conducted in three steps:

1. The content of an attacker-chosen memory location, which is inaccessible to the attacker, is loaded into a register.

2. A transient instruction accesses a cache line based on the secret content of the register.

3. The attacker uses *Flush+Reload* to determine the accessed cache line and hence the secret stored at the chosen memory location.

4.4 How Spectre works

Spectre induces a victim to speculatively perform operations that would not occur during strictly serialized in-order processing of the program's instructions, and which potentially leak victim's confidential information via a covert channel to the adversary. Spectre-type attacks leverage mis-prediction. The most common Spectre attack is the Variant 1 Figure (6) which performs an out-of-bounds array read, exploiting a branch misprediction of the array bounds check.

As it is shown in Figure 5, Spectre attacks are conducted in three steps:

1. The setup phase, in which the processor is mistrained to make an

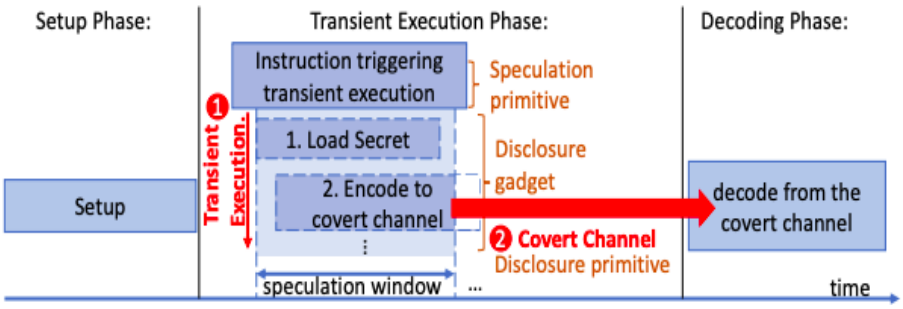


Figure 5: Phases of transient execution attacks

exploitably erroneous speculative prediction. ("*Setup Phase*")

2. The processor speculatively executes instructions from the target context into a microarchitectural covert channel. The piece of code that accesses and transmits secret into the covert channel is called *disclosure gadget*. ("*Transient Execution Phase*")

3. The sensitive data is recovered. This can be done by timing access to memory addresses in the CPU cache. ("*Decoding or Recovering Phase*")

```
uint8 A[10];
uint8 B[256*64];
void victim (size_t addr) {
    if (addr < 10) { // mispredicted branch
M1:     uint8 val = A[addr]; // secret is accessed
M2:     ... = B[64 * val]; // secret is transmitted
    }
}
```

Figure 6: Spectre Variant 1

For example, Spectre Variant 1, shown in Figure 6, bypasses a bounds check due to a branch misprediction and transmits secret data behind that bounds check over a cache-based covert channel. Since the address *addr* can take an arbitrary value, *val* can be any value in program memory, meaning the covert channel can reveal arbitrary program data. First, a secret value is speculatively *accessed* and read into architectural state due to adversary-controlled speculative execution. For instance, load M1 in Figure 3 reads

val even if $addr \geq 10$ due to a branch misprediction, after the adversary has mistrained the branch predictor. Second, that secret value is *transmitted* over a covert channel (formed using one or more younger instructions). For example, load M2 in Figure 6 transmits the secret over a cache-based covert channel.

The sample codes of different variants are shown in Figure 7. Thevictim code should allow a potential mis-prediction or exception to happen. In Spectre V1, to leverage Prediction History Table (PHT) , a conditional branch should exist in the victim code followed by the gadget. Similarly, in Spectre V2 and V5, the victim code should have an indirect jump (or a return from a function) that uses BTB (or RSB) for prediction of the execution path. Specifically in Spectre Variant 2, when the CPU encounters an indirect branch instruction, the branch predictor tries to guess the target address and the CPU immediately starts speculatively executing instructions at this address. The attacker can potentially "poison" the branch target buffer (BTB) to hijack the speculative execution flow and to redirect it to any code location containing gadget instructions. In Spectre V4, to use STL, the victim code should have a store following a load having potential address speculation.

Spectre V1:
The attacker trains the PHT to execute disclosure gadget

```
struct array *arr1 = ...;
struct array *arr2 = ...;
unsigned long offset = ...;
if (offset < arr1_len) {

        sec = arr1[offset];
        value2 = arr2[sec*c];}
```

Spectre V2:
The attacker trains the BTB to jump to the disclosure gadget

```
...
...
...
jmp LEGITMATE_TRGT

TRGT: movzx eax, byte [rdi]
        shl rax, 0Ch;
        mov al, byte [rax+rsi]
```

Spectre V5:
The attacker pollutes the RSB, to return disclosure gadget after Fun1.

```
struct array *arr1 = ...;
struct array *arr2 = ...;
unsigned long offset = ...;
if (offset < arr1_len) {

        sec = arr1[offset];
        value2 = arr2[sec*c];}
```

Spectre V4:
The attacker delays the address calculation causing speculation

```
char * ptr = sec;
char **slow_ptr = *ptr;
clflush(slow_ptr)
*slow_ptr = pub;

        value2 = arr2[(*ptr) *c];
```

Meltdown:
The attacker accesses the address in rcx to cause an exception

```
(rcx = address lead to exception)

Retry:
mov al, byte [rcx]

shl rax, 0xc
jz retry
Mov rbx, qword [rbx + rax]
```

Figure 7: Example codes of Spectre variants and Meltdown

4.5 Spectre & Meltdown Diferrencies

The common theme between Spectre and Meltdown is that they adversely exploit the side effects of transient execution. However, these two attacks exploit different properties of the CPUs. Spectre relies on misprediction events to prompt transient instructions. Spectre works only with data accessible architecturally to an application. In contrast, Meltdown relies on transient out of order instructions following an exception. Meltdown relies on transient instructions inaccessible architecturally to an application. Spectre is considered a harder threat to defeat than Meltdown.

5 Hardware Mitigation Mechanisms

The simplest and most naive way to defend against side channel attacks due to speculation is to not execute speculatively. However, in this way, the in-order processors will eventually come with a huge performance overhead. On account of this, researchers have implemented novel mechanisms for defending against side channel attacks (i.e., Spectre and Meltdown), while exploiting the beneficial effects of speculative execution and maintaining the security. The main idea is to make invisible the side effects of speculation in hardware (i.e., caches, predictors, etc.).

In this diploma thesis, we focus on micro-architectural mitigation techniques and we classify them in three main approaches that make the effects of speculative execution invisible in hardware. These are referred as (i) hiding, (ii) delaying, and (iii) undoing techniques. Next we describe in detail these three approaches. For completeness, we also discuss briefly software mitigation schemes and other approaches that have been proposed to address speculative attacks.

5.1 Hiding the side-effects of speculative execution until speculation is resolved

This approach is taken by solutions such as *InvisiSpec*, *SafeSpec* and *Ghost Loads* [35], [12], [24]. They hide the side effects of transient instructions in specially designed buffers that keep them hidden until the speculation is resolved and the side effects can be made visible. Since these approaches have to wait before they can make the side effects visible, they incur a performance cost relative to how long the side effects need to be hidden.

5.1.1 InvisiSpec

InvisiSpec proposes the principle of "visibility point" of a load, which indicates the time when a load is safe to cause microarchitectural state changes that are visible to the attacker. A "speculative buffer" is used to temporarily cache the load, without modifications in the cache. After the "visibility point" the data will be fetched -securely- into the cache. In addition, for coherency reasons and for being able to handle multi-threaded workloads,

the authors have redesigned the cache coherency policies such that the data needs to be validated when it is fetched from the "speculative buffer". We can also call this approach, a Re-do approach.

InvisiSpec Main Ideas

1) Unsafe Speculative Loads: Any load that performs a read before reaching the head of the ROB, it is considered to be a speculative load. In *InvisiSpec*, the authors are particularly interested in loads that could potentially cause security issues due to speculation. These loads are called Unsafe Speculative Loads (USL). In *InvisiSpec*, Unsafe Speculative Loads depend on the threat model. The authors consider two threat models: Spectre and Futuristic. In the Spectre attack model, USLs are all speculative loads that follow an unresolved control flow instruction (e.g. a branch), while in the Futuristic attack model, every speculative load is considered as Unsafe Speculative Load, except from those speculative loads which are not squashable by an earlier instruction. As soon as a control flow instruction resolves (for Spectre model) or a load becomes non-speculative (for Futuristic model), the load becomes safe.

2) Making USLs Invisible: As mentioned earlier, the main idea of *InvisiSpec* is to make USLs invisible. Thus, any USL would not be able to modify the cache hierarchy or the microarchitectural states in any way that can be visible to the attacker. On top of that, *InvisiSpec* introduces the concept of Speculative Buffer (SB). When a USL is issued, the data are loaded into the Speculative Buffer and not into the local caches. When the load is considered to be safe, *InvisiSpec* re-loads the data, this time into the local caches and makes visible all the side effects of the USL in the memory hierarchy (that is why the "hiding" technique is also called "Re-do" technique).

3) Maintaining Memory Consistency: It is extremely essential to maintain memory consistency, while the USL is issued until it becomes safe and visible to the memory hierarchy (that is called Window of Suppressed Visibility). To solve this issue, *InvisiSpec* may have to perform a validation-step by re-loading the data.

4) Validation or Exposure of a USL: Validation is the way to make visible a USL that would have been squashed during the Window of Suppressed Visibility (due to memory consistency considerations) if, during that window, the core had received an invalidation for the line loaded by the USL. However,

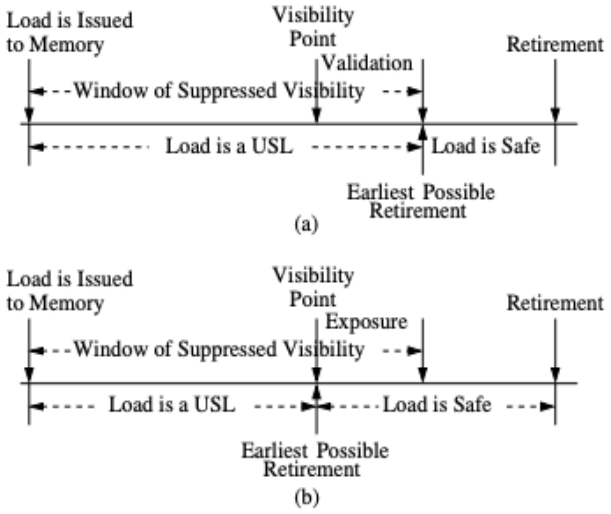


Figure 8: Timeline of a USL with validation (a) and exposure (b)

validations can be expensive. The main reason is that a validation operation includes comparing the actual bytes used by the USL (as stored in the SB) to their most up-to-date value being loaded from the cache hierarchy. Therefore, a USL enduring a validation cannot retire until the transaction finishes, i.e., the line obtained from the cache hierarchy is loaded into the cache. Then, the line's data is compared to the subset of it used in the SB, and a decision regarding squashing is made. Hence, if the USL is at the ROB head and the ROB is full, the pipeline stalls.

InvisiSpec identifies many USLs that could not have violated the memory consistency model during their Window of Suppressed Visibility, and allows them to become visible with a cheap Exposure. These are USLs that would have not been squashed by the memory consistency model during the Window of Suppressed Visibility, if, during that window, the core had received an invalidation for the line loaded by the USL. In an exposure, the line returned by the cache hierarchy is simply stored in the caches without comparison. Hence, the USL does not stall the pipeline.

To summarize, there are two ways to make a USL visible: validation and exposure. The memory consistency model determines which one is needed. Figure 8 shows the timeline of a USL with validation and with exposure.

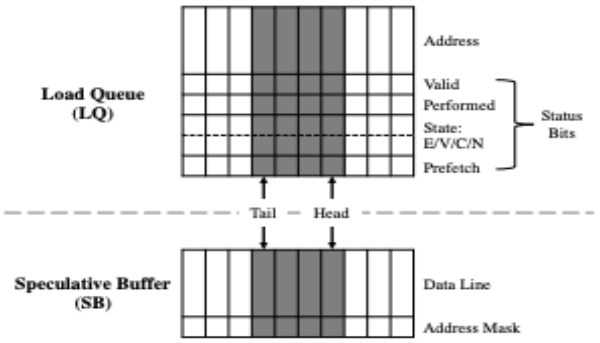


Figure 9: Speculative Buffer and its logical connection to the load queue.

InvisiSpec Design

1) Speculative Buffer in L1. *InvisiSpec* places the Speculative Buffer (SB) close to the core to keep the access latency low. Authors have designed the SB with as many entries as the Load Queue (LQ), and a one-to-one mapping between the LQ and SB entries Figure (9). Therefore, this design makes several operations easy to support. For example, given an LQ entry, *InvisiSpec* can quickly find its corresponding SB entry due to the one-to-one mapping design. Speculative Buffer stores the data of a cache line plus an Address Mask that indicates which bytes were read. The Address Mask is used to compare the data in the SB entry to the incoming data in order to determine whether to validate an SB entry or not. Each LQ entry has some status bits: *Valid*, *Performed*, *State*, and *Prefetch*. *Valid* records whether the entry is valid. *Performed* indicates whether the data requested by the USL has arrived and is stored in the SB entry. *State* indicates the state of the load. It can be "requiring an exposure when it becomes visible" (E), "requiring a validation when it becomes visible" (V), "exposure or validation has completed" (C), and "invisible speculation is not necessary for this load" (N). The latter is used when invisible speculation is not needed, and the access should go directly to the cache hierarchy. Finally, *Prefetch* indicates whether this entry corresponds to a prefetch

2) Supporting Prefetching. *InvisiSpec* supports software prefetch instructions. Such instructions follow the same two steps as a USL. The first step is an "invisible" prefetch that brings a line to the SB without changing any cache hierarchy state, and allows subsequent USLs to access the data locally. The second one is an ordinary prefetch that brings the line to the cache

when the prefetch can be made visible. This second access is an exposure, since prefetches need not go through memory consistency checks. To support software prefetches, *InvisiSpec* increases the size of a core's SB and, consequently, LQ. The new size of each of these structures is equal to the maximum number of loads and prefetches that can be supported by the core at any given time. *InvisiSpec* marks the prefetch entries in the LQ with a set *Prefetch* bit.

3) Per-Core Speculative Buffer in the LLC. InvisiSpec adds a per-core LLC-SB next to the LLC. Its purpose is to store lines that USLs from the owner core have requested from main memory, and to provide the lines when *InvisiSpec* issues the validations or exposures for the same loads. It is a circular buffer with as many entries as the LQ, and a one-to-one mapping between LQ and LLC-SB entries.

4) Securing the D-TLB. To prevent a USL from observably changing the D-TLB state, *InvisiSpec* uses a simple approach. First, on a D-TLB miss, it delays serving it via a page table walk until the USL reaches the point of visibility. If the USL is squashed prior to that point, no page table walk is performed. Second, on a D-TLB hit, any observable TLB state changes such as updating D-TLB replacement state or access/dirty bits are delayed to the USL's point of visibility. It would be possible and would probably offer better performance if *InvisiSpec* used an SB structure like the one used for the caches (like the approach followed in *SafeSpec*).

5.1.2 SafeSpec

SafeSpec [12] is a mitigation mechanism in the same category as *InvisiSpec*. The general principle of *SafeSpec* mechanism is to use temporary structures to store data (referred as shadow states) in order to prevent information leakage due to speculative execution. For instance, if a speculative load instruction causes a load of a cache line, instead of loading that cache into the processor caches, *SafeSpec* mechanism holds the line into the shadow structures. *SafeSpec* is a very similar mechanism to InvisiSpec. Later, if the load is squashed the cache is not updated and the effects are removed leaving no traces. Alternatively, if the instruction commits, the cache line is moved from the temporary structure into the cache and the shadow state is cleared.

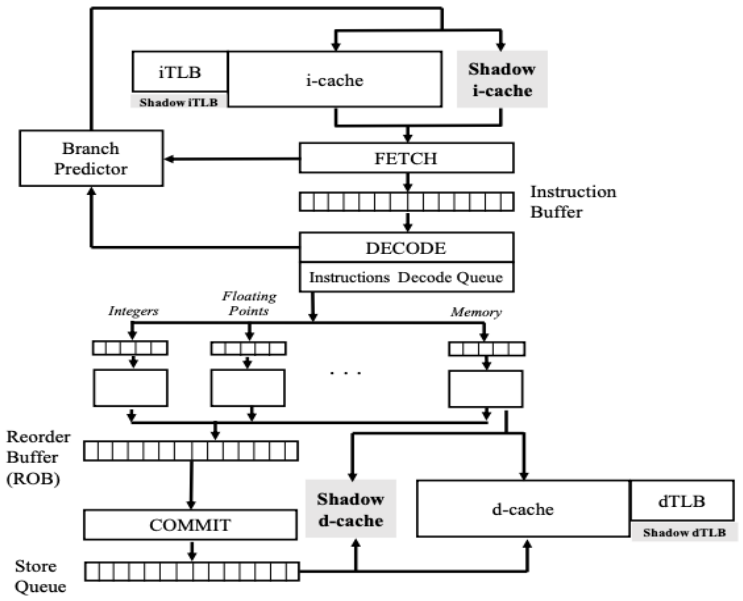


Figure 10: SafeSpec extension to the CPU pipeline

SafeSpec Design

SafeSpec adds shadow states to protect data caches, instruction caches and TLBs (Figure 10). Regarding data caches, which is the most commonly used covert channel, *SafeSpec* adds a shadow structure to hold the cache lines that have been fetched speculatively. It is an associatively-filled lookup table to hold speculatively read cache lines. *SafeSpec* augments the load store queue with a pointer to the shadow cache line for loads operations that are speculative. Any instruction dependent on the speculative load reads the cache line from the shadow structure. Once the load instruction commits, the shadow cache line is written to the caches and freed in the shadow structure. If the load is squashed, the value is just cleared from the shadow structure. If an instruction commits, the cache line is moved from the shadow structure to the caches. If the instruction is squashed, the shadow structure entry is marked as available. In this way, replacement states are not affected or updated by the speculative data that does not commit and thus mis-speculation side-effects remain invisible.

SafeSpec also considers instruction cache as a potential covert channel, albeit it is more difficult to be exploited. In addition, the instruction cache

requires much more effort to protect, since the data dependent branches use the branch predictor, but the I-cache footprint from this branch is not data dependent because the value in the BTB is not data dependent either. Therefore, the authors had to initialize the BTB to a third location and then to introduce sufficient delay in the pipeline for the data dependent branch to be resolved such that it registers the data dependent location in the I-cache.

Finally, *SafeSpec* considers TLBs covert channels as a potential threat of leaking secrets and thus adding shadow structures to protect D-TLBs and I-TLBs. For the instruction cache and the TLBs, SafeSpec creates similar shadow structures and augments the ROB with pointers to the shadow state entries if the instruction is speculative and the cache line (or TLB entry) were fetched speculatively.

In order to provide efficient performance to the processor, the shadow structures should be sized as much as they are needed to accommodate the speculative states. If the shadow buffers are full, some requests simply will be dropped leading to loss of some update and thus more performance overhead. If shadow buffers are sized much more than they are needed, probably the energy and area cost would be tremendous high. Therefore, it is critical to measure effectively the size of each shadow structure. For instance, authors state that for I-cache a shadow structure with about 25 cache lines is sufficient for all of the SPEC2017 benchmarks. In a similar manner for the TLBs, less than 10 entries are sufficient for speculative i-TLB misses, but some benchmarks require more d-TLB entries (up to 25).

5.1.3 Ghost Loads

Another mitigation mechanism, which is also a hiding-based solution, is *Ghost Loads*. The main idea of *Ghost Loads* is to perform speculative loads as uncacheable accesses that do not access the cache state. The authors of this work call these uncacheable accesses as "Ghosts". A "Ghost Load" is a load operation that it is undetectable in the memory hierarchy and specifically in the cache hierarchy.

Ghost Loads have the following characteristics:

1. They are issued like any other memory request.
2. They can hit on any level of the memory hierarchy including private

caches, shared caches, and main memory, in which case the response data are returned directly to the core. The replacement state in the cache remains unchanged.

3. In case of a miss, no cache fills are performed with the response data, and no coherence states are modified.

4. They use a separate set of miss status handling registers (MSHRs) that are not accessible by regular loads. Coalescing between Ghosts is allowed only if they belong to the same context, and so is coalescing Ghosts into in-flight regular loads. Coalescing regular loads into Ghosts is not allowed.

5. Any prefetches caused by Ghost loads are also marked as Ghosts. This assures that an attacker will not be able to train the prefetcher and abuse it as a side-channel.

6. Similarly to the data caches, the relevant translation lookaside buffers (TLBs) are also not updated during the lookups performed by Ghost requests.

Ghost Loads Design

The Ghost Loads mechanism relies on two fundamental concepts: The Ghost Buffer and the Materialization mechanism. In most of the cases, workloads have high percentage of load accesses that are considered as Ghosts. This consideration can lead to a tremendous high performance overhead of the processor. To regain some of that lost performance, the data used by a Ghost load can be installed in the cache after the load is no longer speculative. Materialization (Mtz) is a mechanism for achieving that, by performing all the microarchitectural side-effects of the memory request after the load is no longer speculative. When a load is ready to be committed, an Mtz request is sent to the memory system. The request will act as a regular load request, with the difference that it will not load any data into a CPU register. As such, it will install the cache line into the appropriate caches and update the replacement data. Materialization mechanism needs to be fast as regular accesses to the same cache line follow closely after the Ghosts.

The Ghost Buffer (GhB) is a very small read-only cache that is only accessible by Ghost and Materialization requests. Any data returned by a

Ghost request are placed in the GhB instead of the cache. It is also possible to facilitate prefetching of Ghost requests, by modifying the prefetcher to recognize Ghost requests and tag prefetches initiated by them as Ghosts. The prefetched cache lines can later by installed by the GhB into the cache when the speculation has been resolved.

The point is that while the Ghost Buffer improves by itself the performance of the overall mechanism, it is when combined with the Materialization mechanism that the GhB is thriving. Specifically, when an Mtz request misses in a cache, it then checks the GhB. If the data are found, then they are installed in the cache, eliminating the need to fetch them from somewhere else in the memory hierarchy.

One of the main advantages of *Ghost Loads* over *InvisiSpec* is that the former mitigation scheme can provide safe speculative prefetching. The buffer utilized by *InvisiSpec* has a one-to-one correspondence with the entries of the load queue (LQ). In contrast, the Ghost Buffer functions as a read-only cache that might contain any random set of cache lines. Because of this, Ghost can support prefetching that is triggered by speculative loads, while *InvisiSpec* can only safely prefetch non-speculatively. In an evaluation of contribution of each Ghost mechanism (Ghost Buffer, Materialization, Ghost Prefetching) authors pointed that when the Ghost Buffer and the Materialization are enabled and Ghost Prefetching is disabled there is a 10% performance loss compared to the case when processor operates with Ghost Buffer, Materialization and Ghost Prefetching all enabled. This shows the extreme importance of providing secure speculative prefetching in processor's performance, a feature that was overlooked in *InvisiSpec* mechanism. Finally, *InvisiSpec* does handle the case of TSO memory model (with the trade-off a much more complex design), while Ghost only supports RC model.

5.1.4 Comparison Summary

InvisiSpec only protects from D-cache based attacks and does not support safe speculative prefetching. Similarly, *SafeSpec* adds "shadow buffers" to caches and TLBs, so that speculative changes in caches and TLBs do not happen. However, *SafeSpec* leaves speculative attacks due to multi-threaded workloads out of scope. Finally, *Ghost Loads* method's main advantage is that it can provide protection for secure speculative prefetching.

5.2 Delaying speculative execution until speculation can be resolved

The second class of mitigation techniques refers to delaying-based defenses and follows a different approach than hiding-based solutions. Solutions such as *Delay-on-Miss with Value Prediction* [25], *NDA* [32], *Speculative Taint Tracking (STT)* [39] and *DOLMA* [18] selectively delay instructions when they might be used to leak information. The common theme in all of them is that some speculative instructions are considered unsafe under specific conditions and need to be delayed until the speculation has been resolved. *NDA* and *STT* focus on preventing the propagation of unsafe values at their source, based on the observation that a successful speculative side channel attack consists of two dependent parts, (i) an illegal access instruction (i.e., a speculative load) and (ii) one or more instructions that are dependent to the illegal access and leak the secret ("transmit instructions"). Instead of waking up instructions as soon as the operands are ready, *NDA* wakes up instructions as soon as they are safe. In this way, *NDA* prevents secrets from propagating. In a similar manner, *STT* taints access instructions (instructions that may access secrets, i.e., loads) and untaints them as soon as they are considered safe (i.e., if all operands are untainted). While the execution of load instructions is allowed, the execution of their dependents is delayed.

5.2.1 Speculative Taint Tracking (STT)

Speculative Taint Tracking (STT) [39] selectively restricts the forwarding of certain instructions.

Here we describe in more detail how *STT* works.

1) *Tainting data.* The *STT* framework classifies instructions capable of reading secrets under speculative execution as access instructions. *STT* particularly focuses on load instruction that can be potentially access instructions and taints its output and any dependent instructions on the tainted load.

2) *Untainting data.* The *STT* framework specifies, under certain conditions, when a tainted instruction can be considered safe, and thus to untaint it. The point when a speculative access is no longer considered as a threat, it is referred as Visibility Point. As it is indicated above provided by prior work

in *InvisiSpec*, the visibility point depends on the threat model. In the *Spectre* model, an instruction has reached the visibility point if all older control-flow instructions have resolved. In the *Futuristic* model, an instruction has only reached this point if it cannot be squashed. Instructions before and after the visibility point are called unsafe and safe, respectively, as instructions which have passed the visibility point are not speculative from a security perspective. *STT* untaints the output of an access instruction once it becomes safe.

3) *Classification of instructions that can potentially leak secrets.* The microarchitecture classifies certain instructions as transmit instructions. Covert channels can be explicit or implicit, and implicit channels can be further broken down based on when they leak and their branch type. For instance, to block implicit channels, *STT* requires the microarchitect to classify explicit branch instructions, which affect control-flow, and to identify the implicit branches that represent additional sources of data-dependent resource usage, e.g., store-to-load forwarding, memory consistency speculation

STT Design

The key challenge in the implementation that *STT* addresses and it is worth to mention is how to implement the automatic untaint operation. In general, untainting should be fast. Propagating untaint is non-trivial, because dependency chains can be long and each instruction can have many data dependencies whose taint status needs to be tracked. *STT* addresses these challenges with a novel fast untaint algorithm. The key observation that *STT* makes is that since instructions reach their visibility point in program order to untaint the arguments for an instruction i, it suffices to wait for the youngest access instruction that is causing the taint for i to reach the visibility point. *STT* calls this instruction the *Youngest Root of Taint* (YRoT) of i. Following this approach, *STT* only tracks the position of the YRoT of each instruction in the ROB. Then, it broadcasts the ROB position of each access instruction as it reaches the visibility point and if each younger instruction whose YRoT is smaller or equal to the broadcasted value, it becomes untainted.

STT calculates the Youngest Root of Taint (YRoT) in the processor rename stage. It adds two new fields to the entries in the rename table (which maps logical registers to physical registers): YRoT and the *access instruction ROB index* (AccessInstrIdx), both of which require log2(ROBSize) bits. YRoT tracks the Youngest Root of Taint of the instruction that last produced

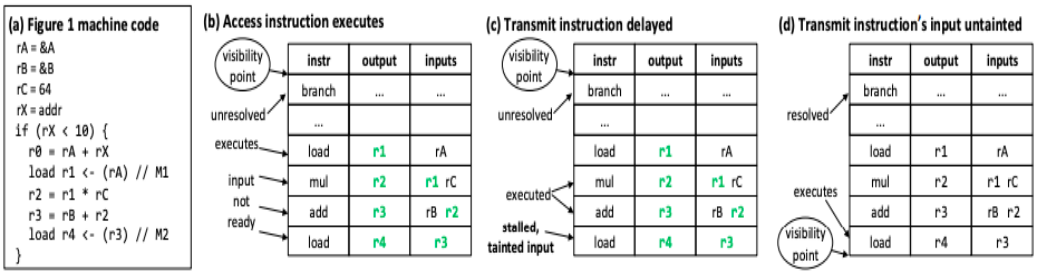


Figure 11: Snapshots of ROB state during the STT execution of the Spectre V1 code, in the Spectre threat model. (Tainted registers are green.)

each logical register in program order. AccessInstrIdx records the ROB index of the last producer for each logical register, if that producer was an access instruction, or -1 otherwise.

STT blocks explicit channels by delaying the execution of any transmit instruction whose operands are tainted until they become untainted. This scheme imposes relatively low overhead because it only delays the execution of transmit instructions if they have tainted operands. For example, a load that only reads a (potential) secret but does not transmit one—such as load M1 in Figure 6— executes without delay. Load M2, however, will be delayed and eventually squashed, thereby defeating the attack.

Figure 8 depicts this scenario in detail. Figure 11(a) shows a sequence of instructions executing the Spectre V1 code; load M1 is an access instruction. Figure 11(b), the access instruction has executed, and its output and all dependencies are tainted. Non-transmit dependent instructions can freely execute, but any transmit dependent instruction like M2 is stalled Figure 11(c). If the speculation succeeds (i.e., rX $<$ 10), the branch resolves as correct and the access instruction becomes safe. In this case, its output becomes untainted and the transmit instruction is allowed to execute Figure 11(d). Although in this example the transmit instruction becomes safe together with the access instruction, this is not true in general (e.g., if there is an unresolved branch between them). Thanks to *STT*'s untaint mechanism, however, even an unsafe transmitter (i.e., that has not reached the visibility point) whose input becomes untainted can execute without having to delay until it reaches the visibility point or head of ROB.

In contrast, if the branch is mispredicted (i.e., rX $\geq$ 10) the transmitter

remains stalled until it is eventually squashed along with the access instructions it depends on

5.2.2 Delay-On-Miss with Value Prediction

The technique Delay-On-Miss with Value Prediction (DoM) [25] is based at the concept of that if we only allow accesses that hit in the L1 data cache to proceed, then we can easily hide any microarchitectural changes until after the speculation has been verified. At the same time, authors propose to prevent stalls by value predicting the loads that miss in the L1. More specifically:

1. For speculative loads that hit in the L1, they are allowed to proceed and use the accessed memory, provided that we do not affect the L1 replacement state or perform any prefetches at that time. This keeps speculative hits invisible.

2. Speculative loads that miss in the L1 level cache are selectively delayed until the speculation has been resolved and value prediction is used instead of sending a request deeper in the memory hierarchy. Value prediction is completely invisible to the outside world, thereby enabling the load to proceed with a value while keeping its speculation invisible. When the load is considered safe, a normal request is issued to the memory system that fetches the actual value. At that point, regardless the value prediction was correct or not, the access is non-speculative and cannot be squashed. It is, therefore, safe to modify the memory hierarchy state.

Delay-On-Miss with Value Prediction Design

For a load marked as "shadowed" (a load under an instruction that can potentially cause a mis-speculation event) the L1 behavior is as follows:

(1) In case of an L1 hit, the cache responds to the request but delays any operations that might cause visible side-effects, such as updating the replacement state. The CPU will signal the cache to perform these operations after the speculation has been successfully verified.

(2) In the case of an L1 miss, the request will simply be dropped. These misses are also referred as shadowed L1 misses.

If a load has received data from the L1 while under a speculative shadow, and after it has left that shadow, it will send a release request to the cache, signaling that the cache can now perform any side-effect causing operations it might have delayed. On the other hand, if a load has not received any data after executing under a speculative shadow, it will simply repeat the initial memory request, this time triggering the normal miss mechanisms in the cache. In case of a cache hit it only needs to delay the side effects of hits, such as updating the replacement data and notifying the prefetcher, which exist outside the critical path of the cache access.

In *DoM with Value Prediction* mechanism, while the number of delayed loads has clearly been diminished, it still incurs high overhead when a L1 miss is encountered. For this reason, authors introduced the concept of Value Prediction on L1 misses in order to make the mechanism faster and more efficient. Value prediction it is used to predict only loads and specifically L1 misses. In this context, value predictors have two interesting properties:

(1) The predictor can be local to the core, isolated from other cores or even other execution contexts. The visible state of the predictor is only updated after the prediction is validated.

(2) Because the predicted value needs to be validated with a normal memory access, the predictor can be incorporated into an OoO pipeline with only small modifications to the L1 cache and no modifications to the remaining memory hierarchy and the coherence protocol.

Authors used the 13-component VTAGE predictor, with 128 entries per component. With the VTAGE predictor authors claim that prediction rate varies significantly between benchmarks workloads and state a value mean of 16 %. However, significant gains can be achieved with this prediction rate if this will enabled along with delay-on-miss.

Comparison with InvisiSpec

Delay On Miss with Value Prediction mechanism outperforms *InvisiSpec* (at least comparing to Futuristic model) showing an average performance overhead of 11% relative to Unsafe Baseline processor. *InvisiSpec* may incur lower overhead for Spectre model (7.6%) but DoM blocks all covert channels due to speculation and not only those related to the cache hierarchy like *InvisiSpec* does. In fact, DoM due to following a different approach of preventing the leaking of information due to mis-speculation by delaying certain

instructions offers a more generic solution. Additionally, it performs quite comparably to *InvisiSpec*, without the hardware complexity cost of modifying the memory hierarchy or the coherence protocol.

5.2.3 NDA

NDA [32] is a hardware mitigation technique used to restrict speculative data propagation in out-of-order processors. *NDA* only allows instruction outputs to flow to dependents if the source instruction is considered safe. *NDA* restricts data propagation by preventing tag broadcast for unsafe instructions, delaying wake-up of their dependants in the issue queue until the source instruction becomes safe.

NDA, in order to defeat Spectre-type attacks, considers any instruction following a predicted branch as unsafe until the branch target and direction are resolved. *NDA* also considers loads that follow a store with an unresolved address as *unsafe*. To mitigate Meltdown-type attacks *NDA* introduces a *propagate-on-retire* mechanism. With this approach, the value returned by **any** load instruction are considered unsafe until the load is ready to retire.

Finally, *NDA's* authors are the first to demonstrate in their work a new type of covert channel that can be exploited even when the cache covert channel is not available—the BTB.

5.2.4 DOLMA

DOLMA [18] offers two protection policies, based on the processor's implementation of speculative execution. *DOLMA-Default* assumes that the processor inherently mitigates all Meltdown-type attacks by preventing potentially faulty micro-ops from broadcasting (i.e., propagating) their results to dependent micro-ops. Therefore, *DOLMA-Default* only addresses Spectre-type attacks. *DOLMA-Conservative* assumes that loads and load-like privileged register reads can transiently bypass exception-like conditions, inducing exception speculation untilthey retire. Thus, in addition to the speculation considerations of *DOLMA-Default*, *DOLMA-Conservative* prevents leakages stemming from all dependants of a load-like micro-op, until the load-like micro-op retires.

DOLMA's key contribution is enforcing a novel principle of transient non-observability that rendering unnecessary the need to delay execution under certain conditions. In addition, *DOLMA* enables protection to scale to registers for a more complete protection with reasonable performance overhead. Transient non-observability is achieved by ensuring that the value of a transient (i.e., destined to squash) operand cannot affect the cycle upon which a non-transient micro-op commits and thus preventing timing-based leakages. More precisely, transient operand values must not cause timing variations in non-transient micro-ops via (a) out-of-order contention for core-local resources, (b) simultaneous uncore/offcore resource access, or (c) persistent state modifications (i.e., modifications that survive the transient window).

Finally, *DOLMA* is the first defense to provide automatic comprehensive protection against existing transient execution attacks for data in both memory and registers.

DOLMA Design

DOLMA novelly applies the technique of "delay-on-miss" to speculative stores, building on prior work that uses delay-on-miss to achieve efficient protection for speculative loads. At a high level, delay-on-miss allows speculative memory micro-ops that hit in first-level core-local structures (e.g., the L1 TLB and—in the case of loads—L1 cache) to execute without stalling until speculation resolves. A speculative memory micro-op that misses in these structures vacates its execution unit and is placed into a dedicated stall queue (as can already be done to mask the latency of TLB misses/-page table walks). Such a design allows other in-flight memory micro-ops to proceed with execution. When speculation resolves, the stalled memory micro-op is re-issued without restriction. Importantly, *DOLMA* ensures that memory micro-ops do not affect replacement policy metadata or memory dependency predictions until speculation resolves, thereby eliminating these potential channels. In addition, if a speculative memory micro-op triggers a prefetch, the prefetch is likewise constrained to delay-on-miss behavior.

At a high level, *DOLMA* adds state to track the speculation status of each micro-op in the re-order buffer (ROB). *DOLMA* then uses this state to delay the execution of instructions, such that transient operands cannot observably affect timing. More specifically, in order to track the speculation status of each micro-op in the pipeline, *DOLMA* conceptually extends each ROB entry with four bits, as shown in Figure 12 : **U**nresolved, **C**ontrol-Dependent,

Data-Dependent, and **P**ending-Redirect. If a micro-op is squashed, the extra bits are ignored.

Unresolved: DOLMA marks an inducive micro-op as unresolved until (a) its associated speculation window resolves, and (b) all elder micro-ops are also resolved. Assuming all elder micro-ops are resolved, a control micro-op resolves when it is executed. Under *DOLMA-Default*, loads are only inducive if they are issued as a result of a hardware prediction unit (e.g., speculative store bypass). Thus, such loads resolve when the corresponding prediction resolves (e.g., the bypassed store executes). Under *DOLMA-Conservative*, all load-like micro-ops are assumed to be unresolved until they retire, in order to handle exception speculation.

Control-Dependent and Data-Dependent: Speculative control dependencies can be easily tracked in *DOLMA*: any micro-op following an unresolved branch in the ROB is control-dependent on that branch, until the next branch introduces a new set of control dependencies. When an unsafe micro-op is issued, a copy of its issue queue entry is placed into a dedicated unsafe queue for in-flight unsafe micro-ops. If an unsafe micro-op executes without stalling, its unsafe queue entry is freed. For unsafe micro-ops that cannot complete for safety reasons, each queue entry holds the index of its youngest unresolved inducer. Such a design allows for efficient wake-up when the micro-op becomes safe. Specifically, if a stalled micro-op's youngest inducer is resolved, the inducer broadcasts its ROB index to this queue such that dependent micro-ops are marked as ready to issue.

Pending-Redirect: Finally, when a frontend-unsafe micro-op would initiate a fetch redirect, its ROB entry is instead marked as pending-redirect. Like backend-unsafe micro-ops, the frontend-unsafe micro-op also vacates its execution unit and awaits a safety broadcast.

Finally, *DOLMA* only clears micro-ops when they become nonspeculative in the context of *DOLMA's* threat models. For control-dependent micro-ops, this means that all elder control-flow micro-ops must be resolved. For data-dependent micro-ops, this means that all elder loads and associated resolvent micro-ops (e.g., stores) must be resolved. When stalled backend-unsafe micro-ops are cleared, they are marked as ready to re-issue from the stall queue.

	micro-ops			DOLMA-Default					DOLMA-Conservative				
a	1	load	r0 -> r1	-	-	-	-		U	-	-	-	U: until retirement
	2	add	r1, r2	-	-	-	-		-	-	D	-	D: until line 1 retires
	3	jump	r1	U	-	-	-	U: until executed/squashed	U	-	D	P	D+P: until line 1 retires
b	1	store	r2 -> r3	-	-	-	-		-	-	-	-	
	2	load	r0 -> r1	U	-	-	-	U: until line 1 executes/squashed	U	-	-	-	U: until retirement
	3	add	r2, r3	-	-	-	-		-	-	-	-	
	4	load	r1 -> r4	-	-	D	-	D: until line 2 resolves	U	-	D	-	D: until line 2 retires
c	1	cmp	0x0, r0	-	-	-	-		-	-	-	-	
	2	jne	r1	U	-	-	-	U: until executed/squashed	U	-	-	-	U: until executed/squashed
	3	load	r2 -> r3	-	C	-	-	C: until line 2 resolves	U	C	-	-	C: until line 2 resolves
	4	load	r4 -> r5	-	C	-	-	C: until line 2 resolves	U	C	-	-	C: until line 2 resolves
	5	jump	r3	U	C	-	P	P: until line 2 resolves	U	C	D	P	D+P: until lines 3 retires

Figure 12: Comparing *DOLMA-Default's* and *DOLMA-Conservative's* handling of speculation status in the ROB in three scenarios. **U** = **U**nresolved, **C** = **C**ontrol-Dependent, **D** = **D**ata-Dependent, and **P** = **P**ending-Redirect. Example (a) shows a non-retired load. Example (b) shows an unresolved speculative store bypass. Example (c) shows an unresolved branch, with a nested branch blocked due to a speculative fetch redirect (line c5)

5.3 Undoing the side-effects of speculative execution in the case of a mis-speculation:

The main idea of this class is to undo any side-effects in a mis-speculation event. The only proposal that falls in this approach is the *CleanupSpec* mitigation mechanism.

CleanupSpec [23] takes a different approach to the previous solutions by permitting speculative execution to proceed unhindered and undoing any side effects in the event of a mis-speculation. When mis-speculation is detected and the pipeline is squashed, the changes to the L1 cache are rolled back. The main cost of the mechanism comes from having to undo the side effects after a mis-speculation.

CleanupSpec Design

CleanupSpec's design approach is to optimize the design for the common-case where loads are correctly speculated. To this end, *CleanupSpec* allows transient loads to speculatively access the cache and make changes as re-

quired. To enable security on a mis-speculation event, are studied the changes a transient load that could make to the data-caches, and delay, reverse or randomize these changes. Thus, *CleanupSpec* is focused on how to undo the side-effects of mis-speculation and in this manner needs to track, protect and reverse these changes on a mis-speculation in order those not to be visible to the attacker.

We briefly outline the main changes to the cache hierarchy made in *CleanupSpec*:

Address Randomization for L2 Cache. To prevent leaks from L2 evictions and replacement policy, address randomization (e.g. CEASER [21]) randomizes the sets that spatially contiguous lines map to, making co-residents of a line in a set unpredictable. As a result, an eviction leaks no information about the address of the Install or L1-Writeback that caused it.

Removing L1 or L2 Installs from the Cache. To prevent a transiently installed line from causing hits on the correct path after a mis-speculation, *CleanupSpec* removes the line from the levels of the cache it got installed in by issuing an invalidation to only those cache levels. This is achieved by tracking which levels of a load caused an install, propagating this information with the load-data through its lifetime in the L1/L2-MSHR and the Load-Queue, till it is retired. This is achieved by tracking the line address of the evicted line in the L1- MSHR on an install and propagating it with the load-data to the Load-Queue. After restoring the evicted lines, it is achieved a L1-cache state such that the unsafe L1-installs and evictions never occurred.

Restoring L1-Evictions. Without randomizing the L1-cache, it is needed to prevent evictions from leaking information. Thus, on a mis-speculation, while removing the installed line, it is also restored the original line that was evicted.

Random Replacement Policy for L1. To prevent replacement state updates on L1-hits from leaking information, it is used random replacement policy for the L1 cache.

Delaying Coherence Changes from M/E to S, till correct path. Only transitions from M/E to S are perceptible due to difference in access-latency and thus these modifications are delayed till the correct path.

5.4 Software Based Defenses

Software mitigations prevent speculative access to secrets by unmapping them (e.g., KAISER [10]) or by disabling speculation in unsafe contexts (e.g.. Retpoline [4], Memory Fences). Unfortunately, many of these mitigations require rewriting Software or OS-changes and are incompatible with legacy code. Recent studies also show that commercially deployed SW mitigations have up to 50% slowdown. In contrast, the proposed hardware mitigation mechanisms have significantly lower overheads and require no software changes.

5.5 Other Mitigation Techniques

At the same time, many secure cache architectures are proposed to use randomization to eliminate cache covert channels in general. For instance, *Random Fill cache* [16] separates the load and the data that is filled into the cache in order the attacker to be unable to disambiguate the sender's access pattern.

Another way to defend against covert channels is to simply degrade the quality of the channel or make it unusable for a practical attack. For instance, many timing covert channels require the receiver to have a fine-grained clock to observe the channel and interpret the results correctly. Therefore, limiting the receiver's observation will eventually reduce the bandwidth or even eliminate the covert channel. Noise can also be added to the channel to reduce the bandwidth and prevent the attacker from extracting sensitive information [27]. However, recently Skarlatos et al. in his paper *MicroScope* [29] demonstrates a new attack class, microarchitectural replay attacks, which are able to denoise nearly arbitrary microarchitectural side channels with only a single run of the victim. Thus, again, all the possible covert channels need to be mitigated to fully mitigate transient execution attacks and provide a robust and secure system.

5.6 Summary

Figure 13 shows in summary the features that every mitigation mechanism protects from. For instance, all defense schemes secure the D-Cache but only

SafeSpec and delay-based solutions protect the instruction cache. In *InvisiSpec*, *Ghost Loads* and *CleanupSpec* mechanisms authors clarify that protection of the I-Cache side channel is feasible but it requires extra amount of work and thus they leave it out of scope. In addition, only *Ghost Loads* and delay-based solution provide secure speculative prefetching, while all the other mitigation mechanism only support software prefetching. *InvisiSpec*, *CleanupSpec* and all delay-based solutions can handle the case of multithreaded workloads but *SafeSpec* and *Ghost Loads* leave this case out of scope as they do not modify the memory hierarchy or the coherence protocols for simplicity. Furthermore, DRAM accesses can also be utilized as a side-channel for attacks. Such attacks are outside the scope of hiding-based or undo-based mechanisms, but they are covered by the delay-based solutions. Finally, most of the solutions provide protection for the DTLB side channel, either by adding extra shadow buffer in the TLB hierarchy (*SafeSpec*, *InvisiSpec*) or restricting the forwarding of the instructions (delay-based). On top of that, *DOLMA*, the current state-of-the-art delay-based solution offers protection for Spectre variants that use speculative stores to transmit the secret, a feature that the previous delay-based solutions were vulnerable.

		Features / Protects from								
		D-Cache	I-Cache	Prefetching	Multithread	Registers (ports?)	DRAM	TLB / stores	BTB	Perf. Overhead
hiding	SafeSpec	✔	✔					✔/ -		
	InvisiSpec	✔			✔	✔		✔/ -		5% - 17%
	Ghost Loads	✔		✔				- / -		12%
delaying	STT	✔	✔	✔	✔	✔	✔	✔/ -		8.7% - 63.4%
	NDA	✔	✔	✔	✔	✔	✔	✔/ -	✔	10.7% - 125%
	DoM w/ VP	✔	✔	✔	✔		✔	✔/ -		11%
	DOLMA	✔	✔	✔	✔	✔	✔	✔/✔	✔	10.2% - 42.2%
undo	CleanupSpec	✔			✔	✔		- / -		5.10%

Figure 13: Table of summarizing the existing mitigation mechanisms that are reviewed in this thesis and the features that they protect from. Prefetching refers that processor provides secure speculative prefetching

6 Methodology

In this section we provide information regarding the experimental methodology we followed to evaluate the various mitigation mechanisms that belong to the three mitigation classes.

6.1 Simulation environment

gem5 [2] is a state-of-the-art cycle-accurate computer simulator, meaning that it simulates hardware components on a cycle-by-cycle basis, instead of simulating the instruction-set of an architecture. It is used both in the academic and industry world for research and development of new hardware technologies. The *gem5* simulator can also execute workloads in a number of ISAs, including today's most common ISAs, x86 and ARM. We use *gem5*

because it meets our requirements to simulate our attack model (cycle accurate), x86 and ARM architecture, caches, out-of-order execution, branch prediction – and is widely used. The authors of the original publications provide open source code implementations[1][2][3][4]

The *gem5* simulator provides a wide variety of capabilities and components which give it a lot of flexibility. These vary in multiple dimensions and cover a wide range of speed/accuracy trade offs as shown in Figure 14. The key dimensions of *gem5*'s capabilities are:

CPU Model. The *gem5* simulator currently provides four different CPU models, each of which lie at a unique point in the speed-vs.-accuracy spectrum. AtomicSimple is a minimal single IPC CPU model, TimingSimple is similar but also simulates the timing of memory references, InOrder is a pipelined, in-order CPU, and O3 is a pipelined, out-of-order CPU model.

System Mode. Each execution-driven CPU model can operate in either of two modes. System-call Emulation (SE) mode avoids the need to model devices or an operating system (OS) by emulating most system-level services.

[1] `https://github.com/efeslab/dolma`
[2] `https://github.com/mjyan0720/InvisiSpec-1.0`
[3] `https://github.com/gururaj-s/cleanupspec`
[4] `https://github.com/cwfletcher/stt`

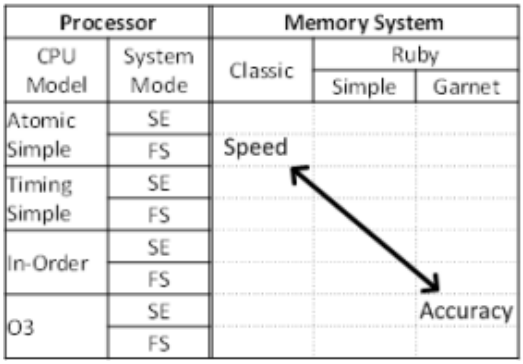


Figure 14: Speed VS Accuracy Spectrum.

Meanwhile, Full-System (FS) mode executes both user-level and kernel-level instructions and models a complete system including the OS and devices.

Memory System. The *gem5* simulator includes two different memory system models, Classic and Ruby. The Classic model provides a fast and easily configurable memory system, while the Ruby model provides a flexible infrastructure capable of accurately simulating a wide variety of cache coherent memory systems.

Table 2 shows the parameters of the simulated architecture that we use in this diploma thesis. When running a SPEC application, we only enable one bank of the shared cache. For SPEC, we used the reference input size, and launch detailed simulation for 1 billion instructions after skipping the first 10 billion instructions. We used the x86 ISA. For both *InvisiSpec* and *CleanupSpec* we used the same configuration setup in gem5, including the Ruby memory model for the unsafe baseline configuration. *DOLMA* and *STT* were not implemented with the Ruby memory model.

Parameter	**Value**
Architecture	1 core (SPEC) at 2.0GHz
Core (OoO)	8-issue, out-of-order, no SMT, 32 Load Queue entries, 32 Store Queue entries, 192 ROB entries, Tournament branch predictor, 4096 BTB entries, 16 RAS entries
Core (in-order)	TimingSimpleCPU from gem5
Private L1-I Cache	32KB, 64B line, 4-way, 1 cycle round-trip (RT) lat., 1 port
Private L1-D Cache	64KB, 64B line, 8-way, 1 cycle RT latency, 3 Rd/Wr ports
Shared L2 Cache	Per core: 2MB bank, 64B line, 16-way, 8 cycles RT local latency, 16 cycles RT remote latency (max)
Network	4×2 mesh, 128b link width, 1 cycle latency per hop
Coherence Protocol	Directory-based MESI protocol
DRAM	RT latency: 50 ns after L2

Table 2: **Parameters of the simulated architecture in gem5**

7 Evaluation

In this section we discuss in detail the evaluation results of one mechanism of each category. In addition we compare the advantages and disadvantages of each mechanism. First, we present and analyze the replicated results of *InvisiSpec* mechanism. For the delaying-based category we have replicated results for both *STT* and *DOLMA* mechanisms. Finally, we discuss the replicated results for the *CleanupSpec* mechanism which represents the undo-based solutions.

7.1 InvisiSpec

Figure 15 compares the execution time of SPEC CPU2006 benchmark applications on 5 different processor configuration schemes. *Base*, is the conventional insecure processor, *Fe-Sp* and *Fe-Fu* puts a fence before every conditional branch/indirect jump or before every load respectively. Finally, *IS-Sp* and *IS-Fu* are the proposed *InvisiSpec* mechanism regarding the Spectre and Futuristic attack model respectively. They are modeled for both RC and TSO memory The results of Figure 15 refer to the TSO memory consistency model. Each set of bars is normalized to *Base*.

If we focus on the fence-based solutions, we clearly see that they suffer from excessive overhead. Under TSO, the average execution time of *Fe-Sp* and *Fe-Fu* is 88% and 246% higher, respectively, than *Base*. In comparison to fence-based solutions, *InvisiSpec* mechanism offers significantly lower performance overhead. Under TSO, the average execution time of *IS-Sp* and *IS-Fu* is 7.6% and 18.2% higher, respectively, than Base.

Evaluating the performance overhead of InvisiSpec mechanism, we can see that the benchmark with the highest performance overhead is *omnetpp*. Replicating the statistics results in gem5 with the same parameters, skipping the first 10 billion instructions and then simulating for the next 1 billion instructions we found that *omnetpp* suffers from around 185% and 200% overhead in IS-Sp and IS-Fu, repsectively. The main reason for this is probably that *omnetpp* is a very TLB-sensitive benchmark and suffers from many TLB misses. *omnetpp* has both a high branch misprediction rate and a high LLC miss rate. In *Base*, TLB misses are not delayed, as they are served speculatively. In contrast, in *IS-Sp* and *IS-Fu*, TLB misses are not served

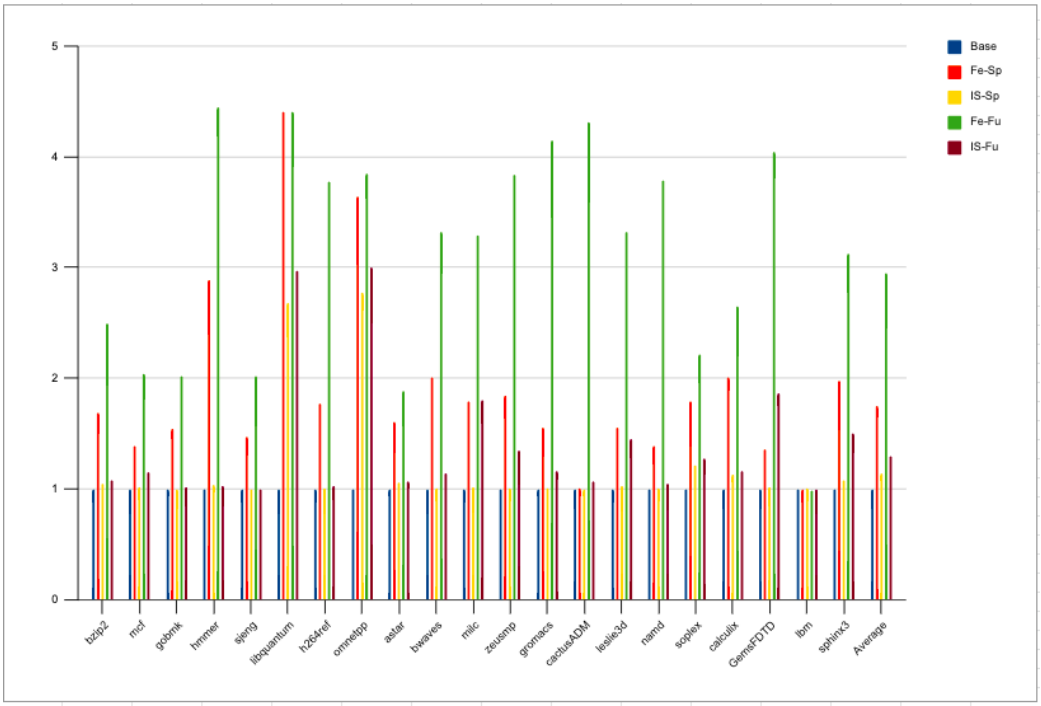


Figure 15: Normalized execution time of the SPEC applications on the 5 different processor configurations.

until the corresponding USL reaches its visibility point. In addition, in case of *libquantum* (both in IS-Sp and IS-Fu) *InvisiSpec* shows high overhead. The main reason for this is that *libquantum* suffers from excessive high L1 D-Cache miss ratio, comparing to the other SPEC CPU2006 benchmarks. Finally, the increase in the performance overhead for *libquantum* is also caused by the fact that *libquantum* is a streaming application that benefits greatly from prefetching, which is disrupted by *InvisiSpec*. On top of that, libquantum also suffers from high validation cost due to large number of LLC misses. However, these results can be often misleading, as when it comes to the actual number of misses and overall memory accesses happening in the application, the absolute number might change while the ratio remains the same.

7.2 STT Results

STT incur up to 63.4% slowdown in processors for full protection of data residing in registers. However, these mechanisms block all covert channels in contrast with the above mechanisms of hiding the side effects of speculation, which only protect from cache based attacks. In comparison to *Delay-on-Miss with Value Prediction*, *NDA* and *STT* only delay transmit instructions, while the former method works on all data. Our evaluated results show only 11% perfomance overhead for Spectre attack model compared to the unsafe baseline processor. The main reason is that only a small portion of all speculative loads (transmitters) are tainted due to older speculative loads (access instructions). For the FUturistic attack model we see 29.7% performance overhead, which differs with the original results of the authors which is wuite close to the performance of the Spectre model. This makes sense because Futuristic is a more restrictive model that forces longer delays before loads can execute.

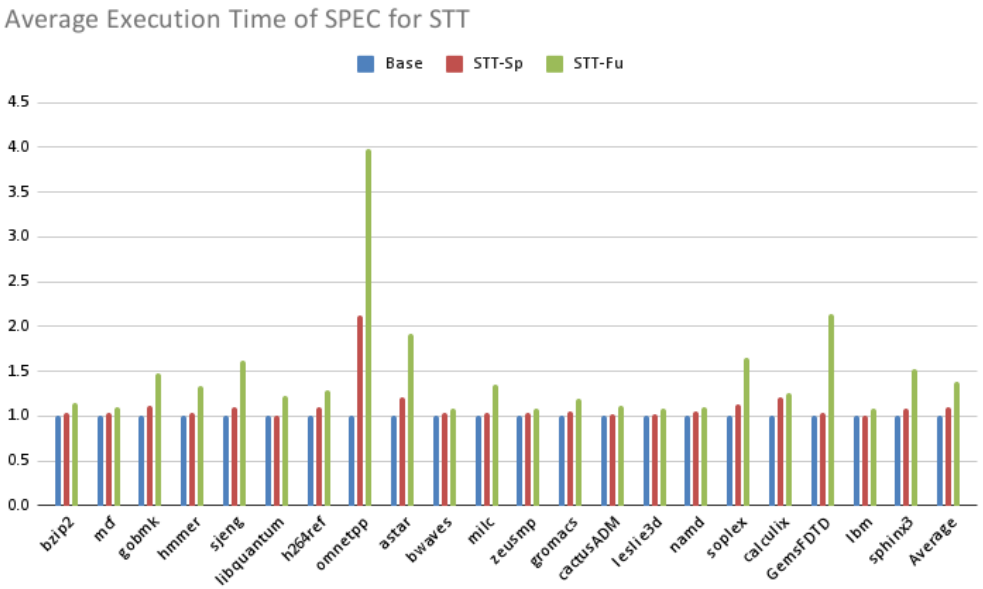


Figure 16: STT single thread performance on SPEC2006.

7.3 DOLMA Results

DOLMA provides protection for both memory-only (M) as well as for memory and registers (M+R). DOLMA-Default (M) in our SPEC2006 evaluation yields 16,8 % performance overhead comparing to the insecure baseline processor. Furthermore, DOLMA-Default (M+R) offers protection for Spectre attacks to both memory and registers by adding 34,9% overhead to the Base processor.

To provide additional protection against Meltdown-type attacks DOLMA-Conservative (M) and (M+R) offer 27,8% and 46,3 % overhead for full protection against all existing Spectre-type and Meltdown-type attacks on data in memory-only and both memory and registers, respectively.

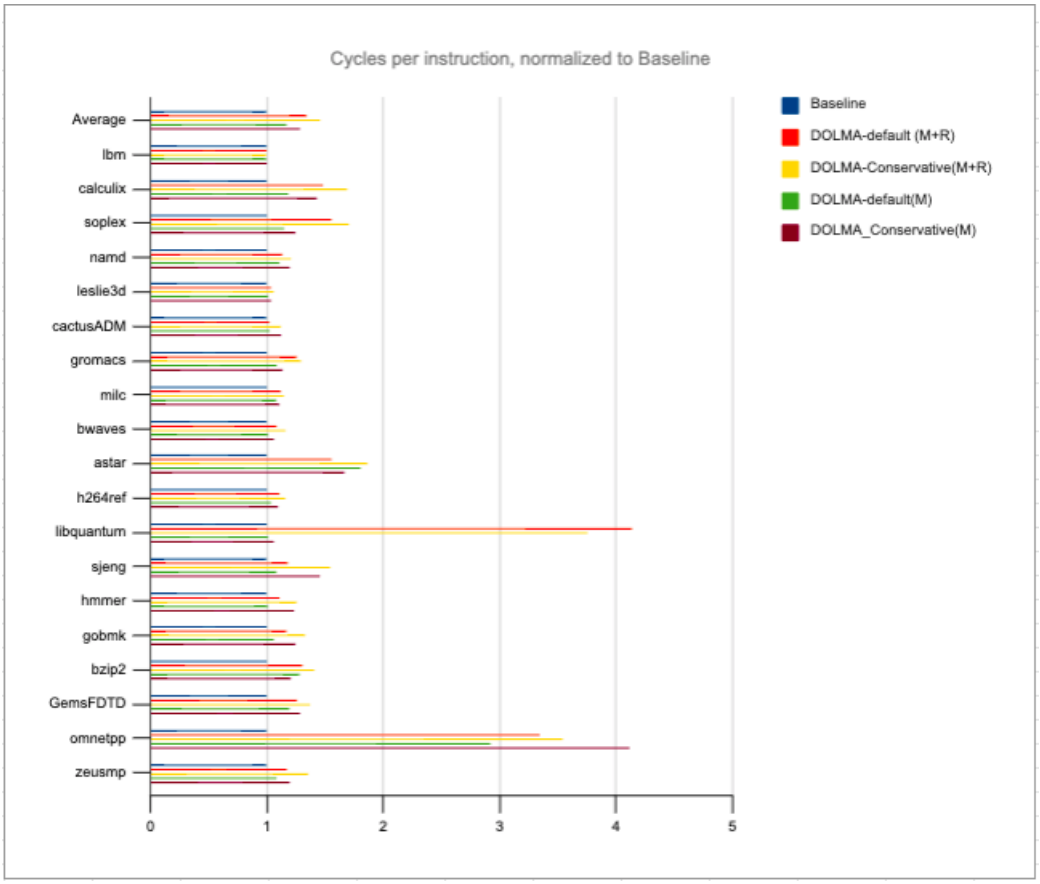


Figure 17: *DOLMA's* single thread performance on SPEC2006.

DOLMA is an improved version of the *NDA* mechanism. *DOLMA* addresses two crucial issues that *NDA* and *STT* fail to resolve. First, *NDA* suffers from very high overhead, especially when it enables protection to registers. Second, *STT* is still vulnerable to arbitrary data leakages through TLB. *DOLMA* significantly improves the performance overhead that *NDA* incurs for protection of data in registers and blocks all covert channels (including the TLB side channel). The method incurs 42.2% performance overhead at worst case scenario (i.e., secret resides in registers), which is 33% faster than *STT*.

7.4 CleanupSpec

Figure 18 shows the execution time of *CleanupSpec*, normalized to that of the Non-Secure baseline. *CleanupSpec* has one configuration scheme which protects from both Spectre-type and Meltdown-type attacks. Thus, it is better to be compared with the Futuristic configuration scheme of *InvisiSpec* and *STT* mitigation mechanisms. The bar "Average" denotes the geometric mean over all the 16 workloads.

In our evaluation, we found that on average *CleanupSpec* incurs a slowdown of 5.9 %. This is because *CleanupSpec* allows the loads to speculatively modify the cache, and incurs no additional overheads for correctly speculated loads. In Figure 13, we notice that benchmarks with the higher branch mis-prediction rates have the highest slowdowns (e.g astar (25%), bzip2 (12%)), whereas benchmarks with lower mis-prediction rates have negligible slowdown (e.g lbm, milc). Furthermore, CleanupSpec incurs higher performance overhead in benchmarks that have higher data cache miss-rates (e.g soplex (7%) , sphinx3 (11%)).

The main cost of CleanupSpec mechanism is due to the cleanup stalls that are in progress when a mis-prediction event is occurred. This stall time depends on the frequency of squashes and the stall-time per squash. It is necessary during a cleanup stall to wait until the correct path loads to complete before the cleanup operations start.

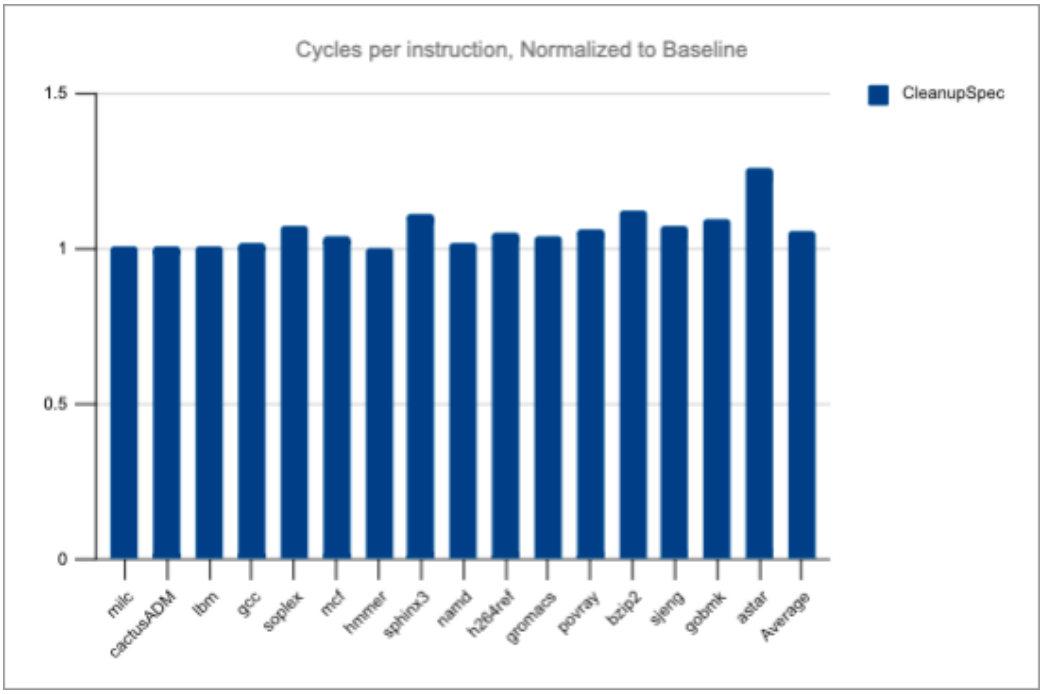


Figure 18: Normalized execution time of the SPEC applications on CleanupSpec design.

7.5 Comparison

STT vs InvisiSpec

These two mitigation mechanisms are difficult to compare because *Invisispec* blocks only covert channels that are related to the cache hierarchy, while *STT* eliminates all covert channels due to the delaying of instructions under certain circumstances. InvisiSpec has exclusively focused on protecting the D-cache. However, this mitigation scheme does not mitigate non D-cache speculative execution attacks [28], [1], [3], [19] [31]. For instance, there has been demonstrated covert transmission of secrets via the instruction-cache (I-cache) [19]. Unfortunately, it is not trivial to apply the same D-cache defense techniques to provide I-cache protection. For example, Sakalis et al. [25] delay speculative loads on an L1 cache-miss to prevent speculative D-cache modifications. However, the authors mention it is difficult to apply the same policy to I-cache misses with low overhead: While d-cache delays

do not preclude other in-flight instructions from executing Out-of-Order, I-cache delays stall the front-end and starve the entire pipeline. Thus, while the authors hypothesize that a similar method could be applied to the I-cache, they do not implement or evaluate the performance overhead of such i-cache protection. In comparison to cache-only defenses, delaying-based solutions (such as *STT, NDA, DOLMA*) are agnostic to the covert channel used in the Transmit Phase and blocks all known attacks. In addition, *STT* does not prevent leaking secrets which are part of the retired state (and thus non-speculative), whereas *InvisiSpec* does handle this case.

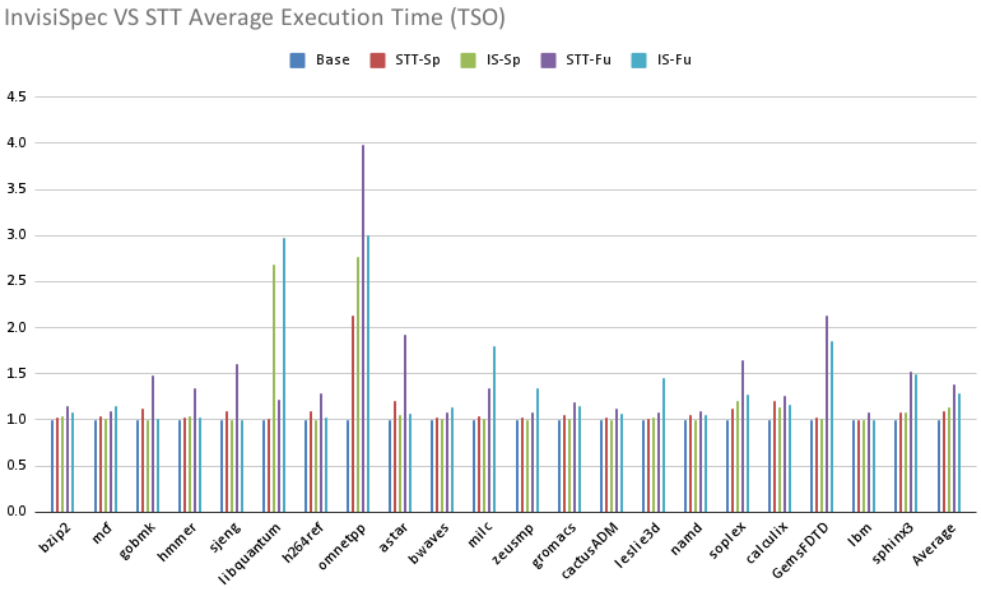


Figure 19: InvisiSpec VS STT average execution time on SPEC2006.

Comparing the overheads of *InvisiSpec* and *STT* using Spectre and Futuristic threat models, we find that *InvisiSpec* and *STT* have 7.6% and 8.5% performance overhead relative to the Unsafe model regarding Spectre model. For Futuristic model *InvisiSpec* and *STT* show 14.5% and 18.2% overhead relative to Unsafe, respectively.

DOLMA vs InvisiSpec

DOLMA, similar to Speculative Taint Tracking and NDA, is based on the

policy of restricting the forwarding of speculative load values. *InvisiSpec* only protects selective load-based transmission channels (e.g the Data-Cache) in contrast to speculative information flow control defenses such as *DOLMA*. While the latter approach might lead to higher performance overhead in some benchmarks than hiding-based solutions, it provides a more overall defense against any covert channel.

DOLMA vs STT

DOLMA's ability to provide protection at lower overhead than *STT* primarily arises from the use of delay-on-miss for memory micro-ops. While *STT* insecurely allows all speculative stores to execute, *STT* conservatively delays all unsafe loads. In contrast, *DOLMA* only delays unsafe loads and stores when they miss in the TLB and—in the case of loads—the L1 cache.

The main difference between these two mitigation mechanisms is that *DOLMA* protects from arbitrary information leakages when a speculative store is used to transmit data through the D-TLB, while *STT* does not cover this case. The reason is that *STT* incorrectly assumes that prohibiting store-triggered speculative cache coherency invalidations is sufficient to prevent transmission via stores in isolation. However, while stores might not speculatively modify cache state on many processors, stores can still leak information via the TLB. As a result of this erroneous assumption,*STT* does not comprehensively prevent transient execution attacks that use stores to transmit a secret-dependent address, whether Spectre-type or Meltdown-type. In this direction, *DOLMA's* contribution is to identify and address another source of leakage via store-to-load forwarding. *STT* does not handle the case of a partial hit (i.e., where a strict subset of the load's address range is found in the store buffer), instead erroneously assuming that the only two possible cases are a complete hit or miss. However, in the case of a partial hit, neither the store buffer nor lower levels of the memory hierarchy hold the correct data in its entirety. Thus, depending on how the microarchitecture handles partial hits, the load may stall until the store completes, revealing information about the store's address via timing. *DOLMA's* approach is to unconditionally issuing the load to the cache hierarchy, and simply ignores the response in the event of an unsafe buffer hit. If the hit was partial (meaning the buffer does not contain all necessary data), the load re-issues once the store is safe and complete. Finally, *DOLMA's* authors have extended the *STT* design in order to support optional protection for

registers (with extra performance overhead). For protection against Spectre-type attacks, *STT* provides *STT-Spectre* configuration scheme. However, unlike *DOLMA-Default*, *STT-Spectre* does not mitigate Spectre-type attacks exploiting data speculation, such as speculative store bypass, nor various transmissions via stores. *STT-Spectre* incurs 11.2% overhead while *DOLMA-default* incurs higher overhead, 17.1%, offering greater protection though. To provide the additional protection against Meltdown-type attacks offered by *DOLMA-Conservative*, *STT-Futuristic* incurs 38.2% overhead but fails to protect select store-based transmissions. In contrast, *DOLMA-Conservative* only incurs 27.9% on data in memory for defeating all existing Spectre-type and Meltdown-type attacks.

DOLMA vs NDA

The main issue of *NDA* mechanism is that suffers from excessive high performance overhead, especially when it comes for enabling protection in registers (125 % for the full protection scheme). The main reason for this is that NDA conservatively prohibits speculative micro-ops from propagating their results to **any** of their dependent micro-ops until speculation resolves. *DOLMA* improves the performance overhead by enforcing the principle of transient non-observability and mitigates all existing transient execution attacks on data in memory and registers.

CleanupSpec vs InvisiSpec

InvisiSpec and *CleanupSpec* are two mitigation schemes that represent two completely different approaches. *InvisiSpec* is a Re-do approach providing safe speculation by issuing the load instruction twice: once to the shadow buffer until the speculation has been resolved and a second time to the cache states updating the side-effects once the load instruction is considered safe. In a completely different manner, *CleanupSpec* mitigation allows the speculation to proceed and does not need to buffer the data and undoes any side-effects in an mis-speculation event.

CleanupSpec outperforms *InvisiSpec*, as incurs less than 6% slowdown in average, nearly three times faster than the slowdown incurred by *InvisiSpec*. The main reason for this is that *CleanupSpec* it requires no extra accesses for correctly speculated loads (which make up the common case), whereas *InvisiSpec* has to re-issue even a correctly speculated to load to the cache hierarchy from the shadow buffers and thus produce higher performance cost.

However, *CleanupSpec* mechanism trades off generality for a simpler but more restrictive solution, since it requires L1 cache replacement policy and randomized cache design. In addition, *CleanupSpec* has focused on protecting the D-cache and does not prevent numerous other covert channels from leaking data during transient execution.

Finally, *CleanupSpec* allows speculative state to propagate through the memory system, using rollback techniques to undo changes to the caches. While this prevents the direct channel from reading the caches once this rollback is complete, the rollback mechanism is itself timing-dependent on secrets brought in by an attacker. Thus, as *CleanupSpec* does not clear speculative state between protection domains, an attacker with code running concurrently with the victim's execution, but before it in program order, can observe state altered by the victim's execution.

Comparison of replicated with original results

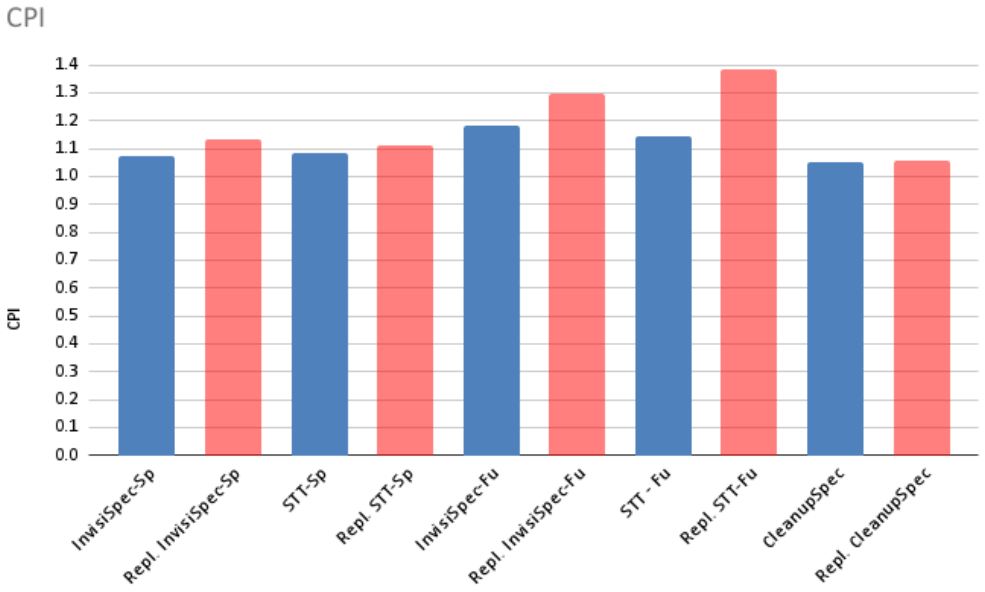


Figure 20: Original and replicated results

In Figure 20, blue and red columns represent the original and our replicated results, respectively. We have replicated the statistic results for one

mechanism of each category. *InvisiSpec* from hiding-based solutions, *STT* from delaying-based solutions and *CleanupSpec* from undo-based solutions. Note that for InvisiSpec the statistic results refer to the TSO memory consistency model. We observe disparities between the original and our replicated results, although we followed the same configuration setup in gem5 simulator as the authors did. While we cannot directly reason about that, one possible explanation is that in our evaluation we excluded some of the benchmarks that the authors used. The mitigation scheme that incurs the highest overhead is *InvisiSpec-Futuristic* in both original and replicated results, while the mechanism with the lower performance overhead is *CleanupSpec*.

8 Conclusions

This diploma thesis provided a survey of attacks due to speculative execution and mitigation techniques. First, in this thesis we explain how the attackers work and can potentially steal sensitive information. We then present most of the existing mitigation mechanisms and categorize them into three main approaches: Hiding-based, Delaying-based and Undoing-based solutions. Finally, we evaluates the performance of one mitigation mechanism from each category and then we discuss and compare with each other mitigation scheme. It is possible to prevent all side-channel attacks in hardware. However, this involves modification of the entire cache hierarchy and this is not feasible. To prevent speculative side-channel attacks, it is possible to modify only the level closest to the CPU, and still achieve strong security properties.

	D-TLB Miss Rate %	D-TLB Misses	Total Delay	Total delay / ticks simulated %	DolmaMisses / D- TLB Misses %
soplex	0.68	2745907	1008958256	0.08	56.6
cactusADM	0.02	111137	1069485676	0.29	28.0
GemsFDTD	1.19	7493237	1619290894	0.18	33.5
omnetpp	24.54	89210801	1026193691	0.04	58.8
astar	2.46	31725503	4127550404	0.32	81.5
zeusmp	0.93	3920471	1034676684	0.22	2.7
bzip2	1.38	9190465	2322630280	0.32	37.1

Figure 21: Total Delay = Hit latency + Miss Latency + Page Fault Latency. DolmaMisses = Misses at L1-DTLB $\rightarrow$ Restricted and placed into a stall queue

9 Future Work

All the proposed mitigation mechanisms have been tested on SPEC CPU2006 workloads in System Emulation (SE) of the gem5 simulator. This is an easy way to evaluate the performance overhead that a mitigation mechanism incurs as the simulation time is low. However, if we look forward to higher precision at the evaluation results we should look to Full System (FS) simulations at gem5. The trade-off would be much longer simulations and many other parameters to set up than in SE simulation.

We devoted considerable amount of time researching how the TLB part affects the overall performance of processor. We noted that in SE mode, all the authors have not modified the gem5's infrastructures to measure the latency of a TLB miss. In general, TLB misses are not modeled in SE mode at all. Although not modeled, they do have an impact on performance for the Out-of-Order processor. Upon a TLB miss, the access is considered as faulty, triggering a pipeline flush and a re-execution. At that point, the SE page table has been populated and the access gets its translation. Therefore, if someone wants to experiment with the impact of the TLB misses in overall performance of processor, it is wiser to run FS simulations.

In our work, we tried to model the TLB latencies at the SE mode for

the *DOLMA* mitigation mechanism. We model 2 clock cycles for the Hit Latency, 50 cycles for the TLB L1 Miss Latency and 2000 cycles for the latency when a page fault occurs. In Figure 21, *DolmaMisses* refer to the misses at the L1-DTLB, which are restricted due to the Delay-on-Miss technique that *DOLMA* applies. We found out that DolmaMisses are a significant fraction of the total D-TLB Misses (6th column). We selectively present benchmarks that are the most TLB-sensitive. For instance, omnetpp, which shows the highest D-TLB Miss Rate (24.5%), more than half of total D-TLB misses (58.8%) are *DolmaMisses*. We expect that the overhead for *DOLMA* mitigation mechanism would be significantly higher if combined with accurate TLB modelling and measuring the real latencies of a TLB miss or page fault. Hence, it is crucial to model the real latencies for the D-TLB in SE mode in order to have more reliable evaluation results in the SE mode simulation.

References

[1] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. CCS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[2] 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. The gem5 simulator. 39(2):1–7, August 2011.

[3] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses, 2019.

[4] Baozi Chen, Qingbo Wu, Yusong Tan, Liu Yang, and Peng Zou. Exploration for software mitigation to spectre attacks of poisoning indirect branches. *IETE Technical Review*, 35(sup1):119–127, 2018.

[5] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, Jun 2019.

[6] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, page 191–206, USA, 2010. IEEE Computer Society.

[7] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. 53(2):693–707, March 2018.

[8] F. Gabbay. Speculative execution based on value prediction research proposal towards the degree of doctor of sciences. 1996.

[9] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks. SEC'18, page 955–972, USA, 2018. USENIX Association.

[10] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. pages 161–176, 06 2017.

[11] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. CCS '16, page 368–379, New York, NY, USA, 2016. Association for Computing Machinery.

[12] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation, 2018.

[13] 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. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[14] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. WOOT'18, page 3, USA, 2018. USENIX Association.

[15] 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. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association.

[16] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 203–215, 2014.

[17] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[18] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing speculation with the principle of transient non-observability. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[19] Andrea Mambretti, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, and Anil Kurmus. Two methods for exploiting speculative control flow hijacks. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, August 2019. USENIX Association.

[20] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. 08 2016.

[21] Moinuddin K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. MICRO-51, page 775–787. IEEE Press, 2018.

[22] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 338–353, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[23] Gururaj Saileshwar and Moinuddin K. Qureshi. Cleanupspec: An "undo" approach to safe speculation. MICRO '52, page 73–86, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Christos Sakalis, Mehdi Alipour, Alberto Ros, Alexandra Jimborean, Stefanos Kaxiras, and Magnus Själander. Ghost loads: What is the cost of invisible speculation? CF '19, page 153–163, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *2019 ACM/IEEE 46th Annual*

International Symposium on Computer Architecture (ISCA), pages 723–735, 2019.

[26] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-leak forwarding: Leaking data on meltdown-resistant cpus (updated and extended version), 2021.

[27] Michael Schwarz, Moritz Lipp, and Daniel Gruss. In *Network and Distributed System Security Symposium 2018*, page 15, February 2018. Network and Distributed System Security Symposium 2018, NDSS'18 ; Conference date: 18-02-2018 Through 21-02-2018.

[28] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network, 2018.

[29] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. Microscope: Enabling microarchitectural replay attacks. ISCA '19, page 318–331, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. SEC'18, page 991–1008, USA, 2018. USENIX Association.

[31] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 178–195, New York, NY, USA, 2018. Association for Computing Machinery.

[32] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. MICRO '52, page 572–586, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Wenjie Xiong and Jakub Szefer. Leaking information through cache lru states, 2020.

[34] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks, 2020.

[35] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 428–441. IEEE Press, 2018.

[36] Y. Yarom, Daniel Genkin, and N. Heninger. Cachebleed: A timing attack on openssl constant time rsa. In *CHES*, 2016.

[37] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. 08 2014.

[38] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. SEC'14, page 719–732, USA, 2014. USENIX Association.

[39] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data. *IEEE Micro*, 40(3):81–90, 2020.