CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

BACHELOR THESIS

Modeling speculative execution over indirect jumps

Autor: Javier López Alonso

Director: Manuel Hermenegildo

Supervisor: Marco Guarnieri

MADRID, JULY 2, 2019

Abstract

Speculative execution is an optimization technique used in modern processors. As illustrated by the recent Spectre attack, attackers can extract sensitive data from the microarchitectural side-effects left behind by speculatively executed instructions.

Spectector is a tool to study these speculative leaks and their countermeasures. In this thesis, Spectector is extended to detect new kinds of leaks produced by speculative execution, more concretely, by branch prediction. This thesis provides the necessary background on the topic, explains the modifications to the Spectector tool, and showcases practical results.

Resumen

La ejecución especulativa es una técnica de optimización presente en los procesadores modernos. Como se demostró en el reciente ataque Spectre, un atacante puede extraer información confidencial a través de los efectos secundarios producidos por instrucciones ejecutadas especulativamente.

Spectector es una herramienta que estudia estas filtraciones causadas por ejecución especulativa y sus contramedidas. En esta tesis, Spectector es extendido para detectar nuevos tipos de filtraciones de información causadas por ejecución especulativa, más concretamente, por predicción de saltos. La tesis proporciona el contexto necesario sobre la materia, explica las modificaciones a Spectector y muestra resultados prácticos.

Contents

1 Introduction

Speculative execution is an optimization technique used by modern processors. It consists in performing tasks before knowing if they will be needed. Processor do this in order to make use of the idle times caused by pipeline stalls. If these computations turn out to be needed, time is saved. Otherwise, the changes performed are reverted and their outcome is ignored. Speculative execution plays a big role in increasing the performance of modern processors. However it also has security implications. When the effects caused by speculatively executed instructions are reverted, they can still leave behind measurable side-effects, which an attacker can exploit to extract sensitive information. An example of this is the recent Spectre vulnerability [1], which was made public on January 2018. Spectre is a family of attacks(with numerous variants) that affects several types of speculative execution. The focus of this thesis is branch prediction. Branch prediction is a type of speculative execution where the processor tries to guess in which way a branch (e.g. `if-else` statement) will go before the branch outcome is known. Spectre affects all modern processors and represents a serious threat. Due to the magnitude of this vulnerability, several countermeasures have been developed at the software level.

The Spectector project's [2] objective is detecting information flows caused by speculatively executed instructions and reasoning about the software counter-measures used by major compilers against Spectre-style attacks. Spectector builds on *speculative-non-interference*, a semantic notion of security against speculative execution attacks. Spectector is implemented in a tool that analyzes x86 assembly programs. It determines if a program presents information leaks, or proves their absence.

The goal of this thesis is to extend Spectector to detect new classes of Spectre-related information leaks. Spectector focuses on conditional branch prediction. However, Spectector's model is limited to speculation over conditional branches.

In this thesis, Spectector's formal model is extended with support for speculative execution over indirect jumps. Indirect jumps are, usually introduced by multi-way branches (`switch` statements) and subroutine calls.

2 Background

Here the relevant background for the rest of the thesis is provided.

2.1 Instruction pipelining

Instruction pipelining is an optimization employed by all modern processors to implement instruction level parallelism into single core processors. Pipelining attemps to keep every part of the processor busy at all times. To do this, the incoming instructions are split into a series of sequential steps (hence the name pipeline) that can be processed by different processing units in parallel. For instance, a classic RISC architecture pipeline has the following stages:

1. Instruction fetch
2. Instruction decode and register fetch
3. Execute
4. Memory access
5. Register write back

Generally speaking, instruction pipelining increases the processor's performance as less processing units are idle while one unit is active(in the case of a non-pipelined processor).

The design of pipelined processors expects instructions to execute in a particular sequence. A branching instruction makes it impossible to know this sequence, as the processor needs to continue executing from another point in the program. This causes delays, known as *pipeline-stalls*, because changing the program execution means that the jump instruction has to pass the execute stage before the next instruction can enter the fetch stage. This introduces large performance losses as the pipeline is empty until the branching instruction is retired from it.

2.2 Branch prediction

Branch prediction is an optimization technique used to reduce pipeline stalls in pipelined processors. It tries to guess the outcome of a branching instruction before it is know definitely (i.e. the branching instruction passes the execution stage) to improve the flow

of the pipeline. It does this by predicting the outcome of the branch and speculatively executing instructions along the predicted path. This means that the processor keeps executing instructions until the correct branch is known. If the prediction was correct, time was saved. However if it was incorrect, all the executed instructions and changes made while executing on the wrong path must be discarded, causing a slight performance drop.

There are several ways of implementing branch predictors in modern processors. Usually the processor keeps a history that tracks whether branches have been taken or not taken. When it encounters a conditional jump that has been several times before it can base the prediction on this history.

2.3 Speculative execution

Speculative execution has been mentioned several times, here I provide a more concise definition. Speculative execution consists, as previously stated, on performing tasks before knowing if they will be needed. This means that the processor *speculatively speculates* on this tasks, instead of just idling, in order to yield a performance gain (if the speculation is correct). When the speculation is correct, the changes are saved, we refer to this as *committing*. If the speculation was incorrect, the effects of the speculation must be discarded, this is known as *rolling-back*.

The security implications of speculative execution are produced by the measurable side-effects after a rollback. To further explain this, the difference between the architecture and the micro-architecture of a CPU must be clear. The architecture is an abstract model of a CPU and depends on the ISA (Instruction Set Architecture, the assembly language). It defines characteristics of a CPU such as the registers, the memory, the flags and, the supported operations. In contrast, the micro-architecture is the actual implementation of the CPU in the hardware, and elements like the cache or the branch predictor state are part of it. When rolling back a speculation, the architectural state is restored. However, the effect of the speculation on the micro-architectural state, specially the contents of the cache lines, is not completely rolled back. This are the side-effects that leak information about sensitive data that Spectre exploits.

2.4 Spectre

After the discovery of the Spectre attack, new attacks exploiting speculative execution have been discovered, but due to the scope of this thesis only Spectre variant 1 and variant 2 will be discussed. Spectre attacks induce a victim to speculatively execute operations that leak confidential information. Spectre v1 targets conditional branch mispredictions,

whereas Spectre v2 targets misprediction of the targets of indirect branches.

The attack begins with a setup phase where the attacker mistrains the branch predictor so that it will later make an exploitable speculative execution. After that, the processor will mispredict, leaving behind side-effects that leave a footprint even after the rollback. Finally the sensitive data can be accessed by the attacker via a covert-channel.

The main example of the Spectre v1 attack relays on leaving a footprint in the state of the cache lines. Then the attacker accesses the sensitive data by timing the access time of the cache lines being monitored. If the access time is short, it means that the cache line was accessed during the speculative execution, if it is long, it means the opposite. By setting up the cache in a specific way beforehand, for instance, by forcing specific data to be out of the cache before the speculation, the attacker can obtain the sensible data.

2.5 Spectector

The objective of Spectector is to analyze programs to detect information leaks caused by speculatively executed instructions or proving them leak-free.

It does this by trying to prove *speculative non-interference*, which is a semantic notion of security against speculative execution attacks. Speculative non-interference compares a computer program's leakages with respect to two different semantics:

- The first is a standard semantics without speculative execution. This semantics is used as a proxy for the intended program behavior.
- The second is a novel, speculative semantics, which can follow mispredicted branches for a bounded number of steps before backtracking. This semantics is used for capturing the effect of speculatively executed instructions.

A program is *speculatively non-interferent* if the speculatively executed instructions do not leak more information into the microarchitectural state than the intended program behavior. Spectector works by symbolically executing all the possible traces of the speculatively semantics and checks for information leaks caused by memory accesses or control flow instructions.

To achieve this, Spectector analyzes x86 programs and translates them to μASM, which is a simple language that captures the main characteristics of assembly languages. The fronted that does this supports a subset of the x86 instruction set. After this, Spectector symbolically executes the translated μASM program with respect to the speculative semantics. To capture the leaks into the μarchitectural state, Spectector adopts an observer

model that sees the location of memory accesses and jump targets from the program execution. The tool then constructs an SMT (satisfiability modulo theories) formula that captures whenever two initial program states produce the same memory access patterns in the standard semantics, as in the speculative semantics. If this formula is valid for every possible symbolic path that the program can take, then the program is speculative non-interferent.

2.5.1 The formal semantics

Spectector intentionally abstracts from several aspects of the architecture and the microarchitecture in their semantics. It only models the meaningful components that leverage speculative execution.

It models the memory and the registers as a pair $\langle m, a \rangle$, called *configuration* σ. m maps memory addresses to values while a maps register identifiers to values. Spectector uses an adversary model that observes the program counter and the memory accesses during computation. This adversary model captures leakage through caches without requiring an explicit cache model.

Since the branch prediction models of modern CPUs are unavailable, Spectector abstracts from branch prediction details using a prediction oracle $\mathcal{O}$. This oracle is queried every time a conditional branch is executed to decide the path of the condition that will be executed speculatively. During analysis the oracle always mispredicts the branch outcomes. This is the worst case scenario, where the leaks are maximized. The oracle also dictates the number of instructions that must be executed speculatively, called *speculative window*. Each time an instruction is speculatively executed, this number is decreased by one.

Spectector models speculative execution as follows. Normal instructions are executed in the standard semantics. When a branching instruction is encountered, the oracle is queried and a snapshot of the current configuration is taken, in case of a rollback. Then a *speculative transaction* begins from the predicted branch. This means that the program is executed speculatively along the predicted branch for a bounded number of computation steps. If the prediction was correct, the transaction is committed and the computation continues with the current configuration. If it was incorrect, the transaction is aborted, the original configuration is restored (using the snapshot that was taken previously) and the computation continues from the correct branch.

The speculative semantics operate on 4 rules:

- SE-NO BRANCH: Captures the behavior of non-branching instructions.

- SE-BRANCH: Models the behavior of branching instructions. It queries the oracle and begins the speculative transaction.
- SE-COMMIT: Captures a speculative transaction's commit. This rule is only applied when the prediction was correct. This is checked by comparing the predicted address with the correct address(obtained from executing the branching instruction non-speculatively).
- SE-ROLLBACK: Captures a speculatively transaction's rollback. It checks whether the prediction was incorrect (by comparing with the non-speculatively executed branching instruction) and restores the previous configuration.

The commit and rollback rules are executed when the speculative window reaches 0.

3 Objective

Spectector's aim is to detect leaks caused by speculative execution over conditional branches. The objective of the thesis is to extend Spectector with speculative execution over indirect jumps. This would allow Spectector to detect new classes of leaks, such as the ones introduced by Spectre v2.

3.1 Indirect jumps

Indirect jumps are a type of program control instruction. Instead of specifying the address of the next instruction to execute, as in direct jumps, the instruction specifies where the address of the next instruction is located. Indirect branches are typically used by programs to implement multiway branches (switch statements), type-dispatched behavior, and subroutine calls.

Speculatively executing indirect jumps consists, as in conditional branches, in trying to predict the outcome of the jump to increase performance. This prediction usually uses a *branch target buffer*. Which is a jump table (a list of program counter location pairs) that stores a target address for each indirect jump.

3.1.1 Leaks from indirect jumps

The leaks originated from speculating over indirect jumps occur in the same way as with conditional branches. After rolling back from a mispredicted indirect jump target, the speculatively executed instructions can leave measurable side-effects in the microarchitectural state.

An attacker could train the indirect branch predictor and then trigger a misprediction where the speculatively executed code leaks information from the victim.

4 Development

4.1 Literature review

The research mainly consisted on obtaining the necessary knowledge to understand the Spectre vulnerability and the Spectector project. The most useful resources were:

- The Spectre paper[1]: This was the paper released to describe the vulnerability named after it. It explains the security implications of speculative execution and shows the different variants of the vulnerability. Some proof-of-concept experiments are discussed and some mitigation options suggested.
- The Spectector paper[2]: This is the paper that explains Spectectors approach to detect Spectre-style attacks. It defines speculative non-interference and their μASM language. It explains in depth the formal semantics used to model the non-speculative and speculative execution of μASM programs and shows real world case studies.
- Spectre is here to stay: An analysis of side-channels and speculative execution [3]. This paper explains microarchitectural side-channels and introduces an architectural model with speculative semantics. It explores software mitigations for information leaks produced by speculative execution's vulnerabilities.

The third paper, released by the Google Zero team, was specially useful. As in Spectector, they define a formal model of a modern CPU, showing that the problems with speculative execution lie at the foundation of this optimization technique. This paper was the starting point to decide how to extend Spectector, as their semantic model is more complete and reflects more elements of a modern CPU, for example, the reorder buffer or the indirect branch predictor.

Lastly, the book *The Formal Semantics of Programming Languages*, by Glynn Winskel was helpul to get familiarized with formal semantics.

4.2 Implementation

The first task after deciding what to add to Spectetor (indirect jumps) was to figure out how to extend the formal semantics that had to be implemented. The approach was to adapt the speculative semantics over conditional branches to indirect branches. The semantics have been extended with:

- a new prediction oracle

- a history for tracking jump instructions targets
- updated speculative semantics rules

The new prediction oracle for indirect jump instructions works with an indirect jump target history. The predictions are based on previously executed jumps. More precisely, the prediction for a jump is the last target address of that jump. Each time a jump instruction is encountered, the branch predictor checks whether there is an entry for that jump in the history:

- If there is an entry, the predictor returns the target address from that entry as the prediction. After that, a new speculative transaction begins.
- If there is no entry for that jump, the jump is executed non-speculatively and a new entry to the history is added.

Before a new speculative transaction begins, the jump history is updated with a new entry. This entry contains the address of the branching instruction and the predicted target address. When a speculative transaction finishes, the jump history is also updated. The new entry contains the address of the jump instruction that started the speculation, and the correct target address for that jump. This is implemented in the adapted speculative semantics. SE-BRANCH queries the predictor and adds entries to the indirect jump history every time an indirect branch instruction is encountered. When a transaction ends, SE-COMMIT or SE-ROLLBACK update the history with the correct result, depending on whether the prediction was correct or not.

5 Results

5.1 Example 1

Here is as example program in the μASM language:

```
0: x<-4
1: jmp(x)
2: load(x,v1)
3: load(y,x)
4: y<-7
```

Listing 1: μASM program

The instructions **0** and **4** represent assignments, the instruction **1** represents and indirect branching instruction, and the instructions **2** and **3**, represent load instructions.

The non-speculative behavior of the program in listing 1 is straightforward. The variable `x` gets assigned the value 4 assigned and then the jump instruction sets the program counter to **4**, where `y` get the value 7 assigned and the program terminates. However, under the speculative semantics the program behaves differently. An attacker could train the branch predictor in such a way that the predicted outcome from `jmp(x)` is **2**. In this case, the load instructions labeled with **2** and **3**, would be executed speculatively, leaving a footprint in the micro-architectural state of the cache.

5.1.1 Spectector's output

This is the output of Spectector, with the predictor mistrained as explained above. As the speculatively executed instructions leak more information into the microarchitectural state than the intended behavior (i.e., what would be leaked under the non-speculatively semantics), Spectector detects that the program is unsafe.

```
program:
  0: x<-4
  1: jmp(x)
  2: load(x,v1)
  3: load(y,x)
  4: y<-7
------------------------------------------------------------------
prg='jmp4.muasm', spec, window_size=200, entry=0, ana=noninter([])
m=[251658240= -1]
a=[pc=0,sp=251658240,bp=15728640]
```

```
[exploring paths]
[path found]
Assignments:
  [A=0,B=0,C=0,D=0]
initial conf:
  i=0
  m=[0xf000000=-1,0=B]
  a=[pc=0,sp=0xf000000,bp=0xf00000,v1=A]
  s=[]
  IJ=[ij(1,-1,2)]
trace:
  0: 1: start(0)
  pc(2)
  2: load(A)
  # element(A,C)
  3: load(C)
  # element(C,D)
  4: 5: rollback(0)
  pc(4)
  4:final conf:
  i=1
  m=[0xf000000=-1,0=B]
  a=[pc=5,sp=0xf000000,bp=0xf00000,x=4,v1=A,y=7]
  s=[]
  IJ=[ij(1,0,4),ij(1,0,2),ij(1,-1,2)]
[checking speculative non-interference]
[path is unsafe,
     showing counter-example initial configurations A and B]
Assignments:
  [A=0,B=0,C=1,D=1]
Case A:
conf:
  m=[0xf000000=-1,0=D,1=C]
  a=[pc=0,sp=0xf000000,bp=0xf00000,v1=A]
trace:
Case B:
conf:
  m=[0xf000000=-1,0=B]
  a=[pc=0,sp=0xf000000,bp=0xf00000,v1=A]
trace:
[program is unsafe]
```

Listing 2: Spectector's output for program in figure 1

Explanation

- `Assignments:` represent placeholders for symbolic variables
- `initial conf:` and `final conf:` show the symbolic configuration, before and after the program execution:
  - `i` represents the identifier for the number of speculative transactions.
  - `m` represents the memory mappings .
  - `a` represents the register values.
  - `s` represents the list of speculative states.
  - `IJ` shows the contents of the branch predictor history. Each entry is represented by a 3-tuple. The first element is the jump instruction address, and the third element is the target destination. The second element is the transaction identifier. In this concrete example the history begins with on entry with an identifier of -1 to represent that the entry was introduced by an attacker.
- `trace:` is the execution trace. It shows that the speculative transaction with id **0** begins at `start(0)` on the instruction labeled with **1** (`jump(x)`) and finishes at `rollback(0)`, with the `load` instructions in between, meaning that they have been executed speculatively.
- Finally, `Case A` and `Case B` showcase two concrete configurations leading to a leak. In this example, the value for `v1` is in both cases **0**. But in configuration A the value obtained from the first `load` is **1**, whereas in configuration B this value is **0**. The second `load` discloses this value. This disclosed value is different between the two configurations, meaning that speculative non-interference does not hold, and therefore proving the leak.

5.2 Example 2

Here, the μASM program is the same one as in figure 1. But this time the predicted outcome for `jmp(x)` sets the program counter to **3**, this means that only one `load` is executed speculatively and therefore no leak is produced, as the leaked value is not disclosed (speculative non-interference holds).

5.2.1 Spectector's output

```
program:
  0: x<-4
  1: jmp(x)
  2: load(x,v1)
  3: load(y,x)
  4: y<-7
-----------------------------------------------------------------
prg='jmp4.muasm', spec, window_size=200, entry=0, ana=noninter([])
m=[251658240= -1]
a=[pc=0,sp=251658240,bp=15728640]
[exploring paths]
[path found]
Assignments:
  [A=0,B=0]
initial conf:
  i=0
  m=[0xf000000=-1,4=A]
  a=[pc=0,sp=0xf000000,bp=0xf00000]
  s=[]
  IJ=[ij(1,-1,3)]
trace:
  0: 1: start(0)
  pc(3)
  3: load(4)
  # element(4,B)
  4: 5: rollback(0)
  pc(4)
  4:final conf:
  i=1
  m=[0xf000000=-1,4=A]
  a=[pc=5,sp=0xf000000,bp=0xf00000,x=4,y=7]
  s=[]
  IJ=[ij(1,0,4),ij(1,0,3),ij(1,-1,3)]
[checking speculative non-interference]
[path is safe]
[concolic] finished, no more conditions to negate
[program is safe]
```

Listing 3: Spectector's output for program in figure 1, different prediction

6 Conclusion

The extended formal model allows Spectector to detect leaks caused by speculation over indirect branches. This can be useful in future work to support detection of Spectre v2 leaks in Spectector.

The semantics can be further extended to model other types of speculation and detect leaks from other Spectre-style attacks. Such as return stack buffers (ret2spec attack) or memory disambiguation predictors (speculative store bypass, SSB).

7 Bibliography

References

[1] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution", *meltdownattack.com*, 2018. [Online]. Available: `https://spectreattack.com/spectre.pdf`.

[2] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: principled detection of speculative information flows", *CoRR*, vol. abs/1812.08639, 2018. arXiv: `1812.08639`. [Online]. Available: `http://arxiv.org/abs/1812.08639`.

[3] R. McIlroy, J. Sevcık, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution", *CoRR*, vol. abs/1902.05178, 2019. arXiv: `1902.05178`. [Online]. Available: `http://arxiv.org/abs/1902.05178`.

8 Appendix

```
%No contents in the ij history, correct prediction
bp_(jmp(E),A,L,N,L,_,IJ, true) :-
  \+ getIJ(~element0(A,pc),IJ,_),
       Ev = ~ev(E,A),
       L = ~concretize(Ev),
  N = 1.

%Get prediction from ij history
bp_(jmp(E),A,L2,N,GoodL2,S, IJ, true) :-
  getIJ(~element0(A,pc),IJ,ij(_, _,L2)),
       Ev = ~ev(E,A),
       GoodL2 = ~concretize(Ev),
  N = ~window(S).
```

Listing 4: Indirect jump predictor

```
% Se-Branch & Se-ExBarrier
xrun1(xc(Ctr,Conf,S, IJ)) := XC2 :-
       enabled(S),
       \+ _ = ~bp(xc(Ctr,Conf,S, IJ)),
       !,
       ( stop(Conf) -> Conf2 = Conf % Note: case for stop/1 (so that spe
       ; trace_rawpc(Conf),
         Conf2 = ~run1(Conf)
       ),
       ( Conf = c(_M,A), spbarr = ~p(~pc(A)) ->
          S2 = ~zeroes(S)
       ; S2 = ~decr(S)
       ),
       XC2 = xc(Ctr,Conf2,S2, IJ).

% Se-Jump
xrun1(xc(Ctr,Conf,S, IJ)) := XC2 :-
       enabled(S),
       trace_rawpc(Conf),
       t(L,N,GoodL, IJFlag) = ~bp(xc(Ctr,Conf,S,IJ)),
       !,
       trace(start(Ctr)), trace_pc(L), track_branch(L),
       Conf = c(M,A),
  (IJFlag = true ->
   IJ2 = [ij(~element0(A,pc),Ctr,L)|IJ]
  ; IJ2 = IJ
  ),
```

```
      Conf2 = c(M,~update0(A,pc,L)), % TODO: it was mset/3 before
      Ctr1 is Ctr + 1,
      XC2 = xc(Ctr1,Conf2,[spec(Ctr,N,L,Conf,GoodL) | S], IJ2).

% Se-Commit
xrun1(xc(Ctr,Conf,S,IJ)) := XC2 :-
  S = [spec(Id,0,L,_ConfPrime,GoodL)|S2],
      enabled(S2),
      L = GoodL,
      !,
      trace_rawpc(Conf),
      trace(commit(Id)),
  (IJ = [ij(IJL,Id,_)|_] ->
  IJ2 = [ij(IJL,Id,GoodL) | IJ]
  ;IJ2 = IJ
  ),
      XC2 = xc(Ctr,Conf,S2, IJ2).

% Se-Rollback-1
xrun1(xc(Ctr,Conf,S, IJ)) := XC2 :-
      S = [spec(Id,0,L,ConfPrime,GoodL)|S2],
      enabled(S2),
      \+ L = GoodL,
      !,
      trace_rawpc(Conf),
      trace(rollback(Id)),
  (getIJbyID(Id,IJ,ij(IJL,Id,_)) ->
   IJ2 = [ij(IJL,Id,GoodL) | IJ]
   ;IJ2=IJ
   ),
      ConfPrime = c(M,A0), A = ~update0(A0,pc,GoodL),
      trace_pc(GoodL), track_branch(GoodL),
      XC2 = xc(Ctr,c(M,A),S2,IJ2).
```

Listing 5: Speculative semantics

Este documento esta firmado por

Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
Fecha/Hora	Tue Jul 02 22:59:47 CEST 2019
Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
Numero de Serie	630
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)