Systematic Analysis of Programming Languages and Their Execution Environments for Spectre Attacks

Amir Naseredini[1,3], Stefan Gast[2,3], Martin Schwarzl[3], Pedro Miguel Sousa Bernardo[4], Amel Smajic[3], Claudio Canella[3], Martin Berger[1,5], Daniel Gruss[2,3]

[1]University of Sussex, UK [2]Lamarr Security Research, Austria
[3]Graz University of Technology, Austria [4]Instituto Superior Técnico, Universidade de Lisboa, Portugal
[5]Turing Core, Huawei 2012 Labs, London, UK

February 09, 2022

Outline

Introduction

The Problem

The Problem

- Spectre mitigations mainly rely on the OS level, or in the execution environment

The Problem

- Spectre mitigations mainly rely on the OS level, or in the execution environment
- We have a large number of mitigations

The Problem

- Spectre mitigations mainly rely on the OS level, or in the execution environment
- We have a large number of mitigations
- We have a vast variety of programming languages with associated execution environments

The Problem

- Spectre mitigations mainly rely on the OS level, or in the execution environment
- We have a large number of mitigations
- We have a vast variety of programming languages with associated execution environments

Problem

It is NOT clear which execution environments have effective mitigations and can securely be used to implement security critical code, and which do not

Our Contributions

Our Contributions

- We systematically analyse the security (with respect to Spectre) of programming languages and their execution environments

Our Contributions

- We systematically analyse the security (with respect to Spectre) of programming languages and their execution environments
- We introduce Speconnector

Our Contributions

- We systematically analyse the security (with respect to Spectre) of programming languages and their execution environments
- We introduce Speconnector
  - It is a novel tool

Our Contributions

- We systematically analyse the security (with respect to Spectre) of programming languages and their execution environments
- We introduce Speconnector
  - It is a novel tool
  - It is to evaluate and exploit Spectre gadgets

Our Contributions

- We systematically analyse the security (with respect to Spectre) of programming languages and their execution environments
- We introduce Speconnector
  - It is a novel tool
  - It is to evaluate and exploit Spectre gadgets
  - It works independent of the target programming language

Our Contributions

- We systematically analyse the security (with respect to Spectre) of programming languages and their execution environments
- We introduce Speconnector
  - It is a novel tool
  - It is to evaluate and exploit Spectre gadgets
  - It works independent of the target programming language
- We demonstrate the security impact with two case studies of security-related libraries, and show that we can leak secrets from them.

Background

Speculative Execution

Speculative Execution

- Programs run conditional branching hence CPUs often do not have a way to choose the next instruction to execute

Speculative Execution

- Programs run conditional branching hence CPUs often do not have a way to choose the next instruction to execute
- With speculative execution, the CPU holds the current state, predict the more probable path based on the history of similar events and speculatively executes in the predicted direction

Speculative Execution

- Programs run conditional branching hence CPUs often do not have a way to choose the next instruction to execute
- With speculative execution, the CPU holds the current state, predict the more probable path based on the history of similar events and speculatively executes in the predicted direction
- If the prediction is not correct the CPU rolls back the architectural state

Speculative Execution

- Programs run conditional branching hence CPUs often do not have a way to choose the next instruction to execute
- With speculative execution, the CPU holds the current state, predict the more probable path based on the history of similar events and speculatively executes in the predicted direction
- If the prediction is not correct the CPU rolls back the architectural state
- HOWEVER, the microarchitectural state is not reverted

Transient-Execution Attacks

Transient-Execution Attacks

- Since the microarchitectural state is not reverted the effects of transient instructions can be reconstructed on the architectural level

Transient-Execution Attacks

- Since the microarchitectural state is not reverted the effects of transient instructions can be reconstructed on the architectural level
- Attacks of this type traditionally use side-channel attacks to reconstruct the architectural state

Gadgets

Definition

A gadget is a piece of code used to transfer the secret information from the victim's side into a covert channel from which the attacker can then retrieve it

Gadgets

Definition

A gadget is a piece of code used to transfer the secret information from the victim's side into a covert channel from which the attacker can then retrieve it

Here is an example of an index gadget

Gadgets

Definition

A gadget is a piece of code used to transfer the secret information from the victim's side into a covert channel from which the attacker can then retrieve it

Here is an example of an index gadget

$\Downarrow$

Gadgets

Definition

A gadget is a piece of code used to transfer the secret information from the victim's side into a covert channel from which the attacker can then retrieve it

Here is an example of an index gadget

⇓

Example

```
if(x < length_of_data){
  tmp &= lookup_table[data[x] << 12];
}
```

Program Execution

- We categorize the execution environments into three categories based on the program execution

Program Execution

- We categorize the execution environments into three categories based on the program execution
  - Interpreted Program Execution

Program Execution

- We categorize the execution environments into three categories based on the program execution
  - Interpreted Program Execution
  - Compiled Program Execution

Program Execution

- We categorize the execution environments into three categories based on the program execution
  - Interpreted Program Execution
  - Compiled Program Execution
  - Managed Program Execution

Program Execution

- We categorize the execution environments into three categories based on the program execution
  - Interpreted Program Execution
  - Compiled Program Execution
  - Managed Program Execution

Note!

This distinction is orthogonal to programming language choice since every language can be interpreted, compiled, and executed in hybrids.

Interpreted Program Execution

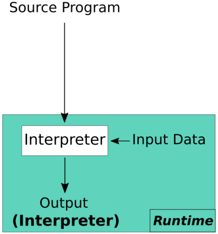
- Interpreted languages need to be translated every time they are being run

Interpreted Program Execution

- Interpreted languages need to be translated every time they are being run
  - Therefore they are more portable as only the interpreter is platform specific

Interpreted Program Execution

- Interpreted languages need to be translated every time they are being run
    - Therefore they are more portable as only the interpreter is platform specific



Compiled Program Execution

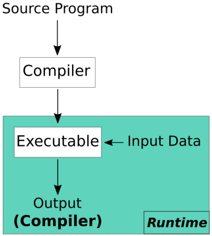
- Compiled languages only incur the overhead of translating the code once

Compiled Program Execution

- Compiled languages only incur the overhead of translating the code once
    - Therefore compilers can perform more sophisticated optimisations since their translation time is less important

Compiled Program Execution

- Compiled languages only incur the overhead of translating the code once
  - Therefore compilers can perform more sophisticated optimisations since their translation time is less important

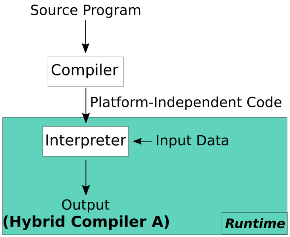


Managed Program Execution

- The aim is to combine the advantages of compiled and interpreted languages

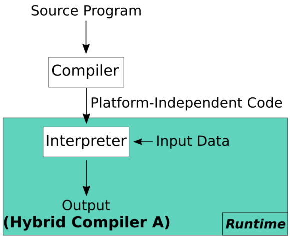
Managed Program Execution

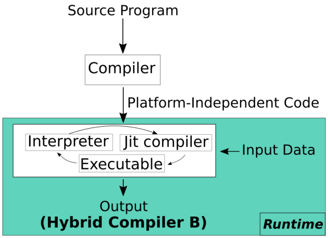
- The aim is to combine the advantages of compiled and interpreted languages



Managed Program Execution

- The aim is to combine the advantages of compiled and interpreted languages





Feasibility of Attacks in Documentations

Interpreted Languages

Interpreted Languages

- We studied 9 different interpreters

Interpreted Languages

- We studied 9 different interpreters
- We looked into the publicly available documentation of each case

Interpreted Languages

- We studied 9 different interpreters
- We looked into the publicly available documentation of each case
- As an additional source of information, we contacted developers of the respective interpreters

Interpreted Languages

- We studied 9 different interpreters
- We looked into the publicly available documentation of each case
- As an additional source of information, we contacted developers of the respective interpreters
- Unfortunately, this step did not provide any additional insights for 8 of them

Interpreted Languages

- We studied 9 different interpreters
- We looked into the publicly available documentation of each case
- As an additional source of information, we contacted developers of the respective interpreters
- Unfortunately, this step did not provide any additional insights for 8 of them

Attack \ PLs	Ruby (MRI)	PHP	Shell (Bash)	Perl	PowerShell (pwsh)	TSQL	Lua	Vim script	Emacs Lisp
Spectre-PHT	×	×	×	☒	×	×	×	×	×
Spectre-BTB	×	×	×	☒	×	×	×	×	×
Spectre-RSB	×	×	×	☒	×	×	×	×	×
Spectre-STL	×	×	×	☒	×	×	×	×	×

Compiled Languages

Compiled Languages

- We considered 15 different compilers in our study

Compiled Languages

- We considered 15 different compilers in our study
- We followed the same approach as the previous part

Compiled Languages

- We considered 15 different compilers in our study
- We followed the same approach as the previous part
- Based on our analysis, the Go compiler has the best situation regarding its mitigations against differenct Spectre variants

Compiled Languages

- We considered 15 different compilers in our study
- We followed the same approach as the previous part
- Based on our analysis, the Go compiler has the best situation regarding its mitigations against differenct Spectre variants

Attack \ PLs	Go	C++ (GCC)	C++ (MS)	C++ (Intel)	C++ (LLVM)	C (GCC)	C (MS)	C (Intel)	C (LLVM)	Rust (LLVM)	Swift (LLVM)	DM	Objective-C (LLVM)	Haskell (GHC)	OCaml (ocamlopt)
Spectre-PHT	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	×	☑	×	⊠
Spectre-BTB	☑	☑	⊠	☑	☑	☑	⊠	☑	☑	☑	☑	×	☑	×	⊠
Spectre-RSB	☑	☑	⊠	⊠	×	☑	⊠	⊠	×	×	×	×	×	×	⊠
Spectre-STL	⊠	×	☑	⊠	×	×	☑	⊠	×	×	×	×	×	×	⊠

Managed Languages

Managed Languages

- We analysed 13 different programming languages and their 18 respective hybrid compilers

Managed Languages

- We analysed 13 different programming languages and their 18 respective hybrid compilers
- We followed the same approach as the previous parts

Managed Languages

- We analysed 13 different programming languages and their 18 respective hybrid compilers
- We followed the same approach as the previous parts
- Surprisingly, the majoity of them either
  - Did not have any mitigations implemented OR
  - Did not provide any information about implemented mitigations publicly

Managed Languages

- We analysed 13 different programming languages and their 18 respective hybrid compilers
- We followed the same approach as the previous parts
- Surprisingly, the majoity of them either
  - Did not have any mitigations implemented OR
  - Did not provide any information about implemented mitigations publicly

Attack \ PLs	Dart	Java (OracleJDK)	Java (OpenJDK)	Java (GraalVM)	JavaScript (SpiderMonkey)	JavaScript (V8)	JavaScript (Chakra)	TypeScript	CoffeeScript	Python (PyPy)	Scala	C#	Elixir	Clojure	Python (CPython)	OCaml (ocamlc/ocamlrun)	Kotlin	Groovy
Spectre-PHT	×	×	⊠	☑	☑	☑	☑	×	×	⊠	⊠	⊠	⊠	×	×	⊠	⊠	⊠
Spectre-BTB	×	×	⊠	×	☑	☑	☑	×	×	⊠	⊠	⊠	⊠	×	×	⊠	⊠	⊠
Spectre-RSB	×	×	⊠	×	×	×	×	×	×	⊠	⊠	⊠	⊠	×	×	⊠	⊠	⊠
Spectre-STL	×	×	⊠	×	×	⊠	×	×	×	⊠	⊠	⊠	⊠	×	×	⊠	⊠	⊠

Speconnector

Threat Model

Threat Model

- Regular Spectre attack threat model

Threat Model

- Regular Spectre attack threat model
  - The attacker is a co-located program running under the same operating system

Threat Model

- Regular Spectre attack threat model
  - The attacker is a co-located program running under the same operating system
  - The attacker is able to execute arbitrary code on the victim machine

Threat Model

- Regular Spectre attack threat model
  - The attacker is a co-located program running under the same operating system
  - The attacker is able to execute arbitrary code on the victim machine
  - The victim code has an interface that we can interact with

Threat Model

- Regular Spectre attack threat model
  - The attacker is a co-located program running under the same operating system
  - The attacker is able to execute arbitrary code on the victim machine
  - The victim code has an interface that we can interact with
- The possibility of an attack happening depends on whether the victim leaks

Threat Model

- Regular Spectre attack threat model
  - The attacker is a co-located program running under the same operating system
  - The attacker is able to execute arbitrary code on the victim machine
  - The victim code has an interface that we can interact with
- The possibility of an attack happening depends on whether the victim leaks
  - Therefore, we focus on the illegal data leakage

Threat Model

- Regular Spectre attack threat model
  - The attacker is a co-located program running under the same operating system
  - The attacker is able to execute arbitrary code on the victim machine
  - The victim code has an interface that we can interact with
- The possibility of an attack happening depends on whether the victim leaks
  - Therefore, we focus on the illegal data leakage
  - We use Speconnector to measure and verify this leakage

Threat Model

- Regular Spectre attack threat model
  - The attacker is a co-located program running under the same operating system
  - The attacker is able to execute arbitrary code on the victim machine
  - The victim code has an interface that we can interact with
- The possibility of an attack happening depends on whether the victim leaks
  - Therefore, we focus on the illegal data leakage
  - We use Speconnector to measure and verify this leakage

Note!

Note that this shows that an attack is possible, and crafting a concrete end-to-end exploit for each language only requires further engineering steps

Method

Method

- The target code first allocates 256 pages of memory

Method

- The target code first allocates 256 pages of memory
- The target code fills the allocated memory with a known magic value

Method

- The target code first allocates 256 pages of memory
- The target code fills the allocated memory with a known magic value
- Speconnector also allocates the same amount of memory

Method

- The target code first allocates 256 pages of memory
- The target code fills the allocated memory with a known magic value
- Speconnector also allocates the same amount of memory
- Speconnector uses the information of the process of the target code to scan for the pages that contain the magic value

Method

- The target code first allocates 256 pages of memory
- The target code fills the allocated memory with a known magic value
- Speconnector also allocates the same amount of memory
- Speconnector uses the information of the process of the target code to scan for the pages that contain the magic value
- Speconnector establishes shared memory between the two processes

Method

- The target code first allocates 256 pages of memory
- The target code fills the allocated memory with a known magic value
- Speconnector also allocates the same amount of memory
- Speconnector uses the information of the process of the target code to scan for the pages that contain the magic value
- Speconnector establishes shared memory between the two processes
- Any victim accesses to one of the now shared pages results in a cache hit and Speconnector catches it by performing *Flush + Reload*

Feasibility of Attacks in Practice

Interpreted Languages

Interpreted Languages

- We were able to exploit one interpreter

Interpreted Languages

- We were able to exploit one interpreter
  - Perl

Interpreted Languages

- We were able to exploit one interpreter
  - Perl
- A potential explanation for all the other interpreters is that

Interpreted Languages

- We were able to exploit one interpreter
  - Perl
- A potential explanation for all the other interpreters is that
  - The speculation window might have been too small for them to fit the attack in it

Interpreted Languages

- We were able to exploit one interpreter
  - Perl
- A potential explanation for all the other interpreters is that
  - The speculation window might have been too small for them to fit the attack in it

Attack \ PLs	Emacs Lisp	Ruby (MRI)	PHP	Shell (Bash)	Perl	PowerShell (pwsh)	TSQL	Lua	Vim script
Depends on setting	-	-	-	-	-	-	-	-	-
Covert Channel	✓	✓	✓	×	✓	✓	×	✓	×
Spectre Attack	×	×	×	×	✓	×	×	×	×

Compiled Languages

Compiled Languages

- We were able to establish a covert channel in 14 out of 15 compilers

Compiled Languages

- We were able to establish a covert channel in 14 out of 15 compilers
- And 12 of them were generating a code that is vulnerable against at least one varient of Spectre attack

Compiled Languages

- We were able to establish a covert channel in 14 out of 15 compilers
- And 12 of them were generating a code that is vulnerable against at least one varient of Spectre attack

Attack \ PLs	Go	C++ (GCC)	C++ (MS)	C++ (Intel)	C++ (LLVM)	C (GCC)	C (MS)	C (Intel)	C (LLVM)	Rust (LLVM)	Swift (LLVM)	DM	Objective-C (LLVM)	Haskell (GHC)	OCaml (ocamlopt)
Depends on setting	*	*	*	*	*	*	*	*	*	*	*	-	*	-	-
Covert Channel	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	✓
Spectre Attack	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	×	✓

Managed Languages

Managed Languages

- We were able to demonstrate a functioning covert channel in 100% of managed languages

Managed Languages

- We were able to demonstrate a functioning covert channel in 100% of managed languages
- We introduced attacks for compilers that were so far not known to be vulnerable, i.e., no Spectre attack on these has been demonstrated before

Managed Languages

- We were able to demonstrate a functioning covert channel in 100% of managed languages
- We introduced attacks for compilers that were so far not known to be vulnerable, i.e., no Spectre attack on these has been demonstrated before
  - It includes Dart, Java, C#, Scala, Groovy, Kotlin and OCaml (ocamlc/ocamlrun)

Managed Languages

- We were able to demonstrate a functioning covert channel in 100% of managed languages
- We introduced attacks for compilers that were so far not known to be vulnerable, i.e., no Spectre attack on these has been demonstrated before
  - It includes Dart, Java, C#, Scala, Groovy, Kotlin and OCaml (ocamlc/ocamlrun)

Attack \ PLs	Dart	Java (OracleJDK)	Java (OpenJDK)	Java (GraalVM)	JavaScript (SpiderMonkey)	JavaScript (V8)	JavaScript (Chakra)	TypeScript	CoffeeScript	Python (PyPy)	C#	Scala	Elixir	Clojure	Python (CPython)	OCaml (ocamlc/ocamlrun)	Kotlin	Groovy
Depends on setting	-	-	-	*	*	*	*	-	-	-	-	-	-	-	-	-	-	-
Covert Channel	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Spectre Attack	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	×	×	×	✓	✓	✓

Case Studies

Case Studies

Case Studies

- We take two case stdies to demonstrate how a Spectre attack can be used to leak secret information from real-world libraries

Case Studies

- We take two case stdies to demonstrate how a Spectre attack can be used to leak secret information from real-world libraries
- These two are:

Case Studies

- We take two case stdies to demonstrate how a Spectre attack can be used to leak secret information from real-world libraries
- These two are:
  - Alice, which is a libray written in Java

Case Studies

- We take two case stdies to demonstrate how a Spectre attack can be used to leak secret information from real-world libraries
- These two are:
  - Alice, which is a libray written in Java
  - cryptokit, which is a libray written in OCaml

Case Studies

- We take two case stdies to demonstrate how a Spectre attack can be used to leak secret information from real-world libraries
- These two are:
  - Alice, which is a libray written in Java
  - cryptokit, which is a libray written in OCaml
- We also argue that a mitigation at the compiler level prevents our attacks

Case Studies

- We take two case stdies to demonstrate how a Spectre attack can be used to leak secret information from real-world libraries
- These two are:
  - Alice, which is a libray written in Java
  - cryptokit, which is a libray written in OCaml
- We also argue that a mitigation at the compiler level prevents our attacks

Note!

Both case studies are using the vulnerable programming languages demonsterated in Section Feasibility of Attacks in Practice of this presentation

Conclusion

Conclusion

Conclusion

- We did a systematic analysis of different programming languages and their respective compilers/interpreters against Spectre

Conclusion

- We did a systematic analysis of different programming languages and their respective compilers/interpreters against Spectre
- We analysed them in theory and practice

Conclusion

- We did a systematic analysis of different programming languages and their respective compilers/interpreters against Spectre
- We analysed them in theory and practice
- We introduced Speconnector

Conclusion

- We did a systematic analysis of different programming languages and their respective compilers/interpreters against Spectre
- We analysed them in theory and practice
- We introduced Speconnector
- We showed Spectre attacks in 8 programming languages not investigated so far and not known to be vulnerable

Conclusion

- We did a systematic analysis of different programming languages and their respective compilers/interpreters against Spectre
- We analysed them in theory and practice
- We introduced Speconnector
- We showed Spectre attacks in 8 programming languages not investigated so far and not known to be vulnerable
- We illustrated the security impact of our results using two case studies

Thank you for your attention