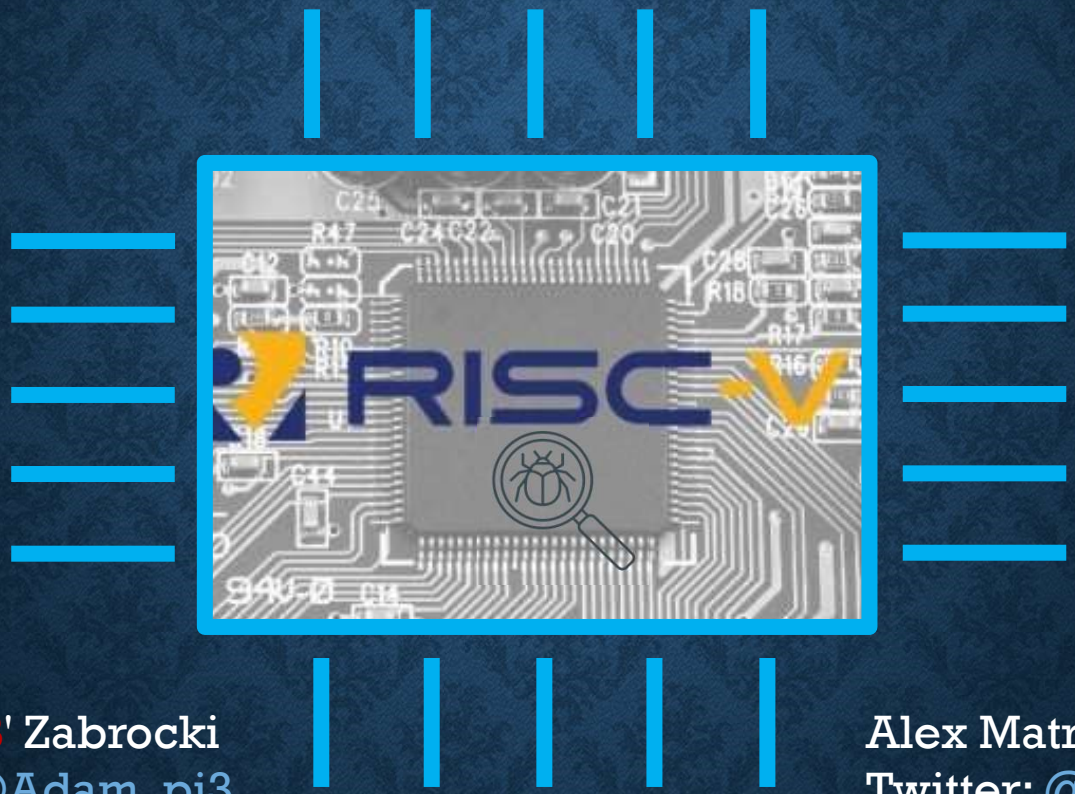
GLITCHING RISC-V CHIPS:



Adam 'pi3' Zabrocki
Twitter: @Adam_pi3

Alex Matrosov
Twitter: @matrosov

MTVEC CORRUPTION FOR HARDENING ISA

/USR/BIN/WHOWEARE

Private contact:

http://pi3.com.pl
pi3@pi3.com.pl
Twitter: @Adam_pi3

Adam 'pi3' Zabrocki:

- Phrack author
- Bughunter (Hyper-V, Intel/NVIDIA vGPU, Linux kernel, OpenSSH, Apache, gcc SSP / ProPolice, Apache, xpdf, more…) – CVEs
- The ERESI Reverse Engineering Software Interface
- Creator and a developer of Linux Kernel Runtime Guard (LKRG)
- More…

Private contact:

github.com/binarly-io
Twitter: @matrosov

Alex Matrosov:

- Security REsearcher since 1997
- Conference speaker and trainer
- Breaking all shades of firmware
- codeXplorer & efiXplorer IDA plugins
- Author "Bootkits and Rootkits" book
- Founder of Binarly, Inc.
- More...

NVIDIA®

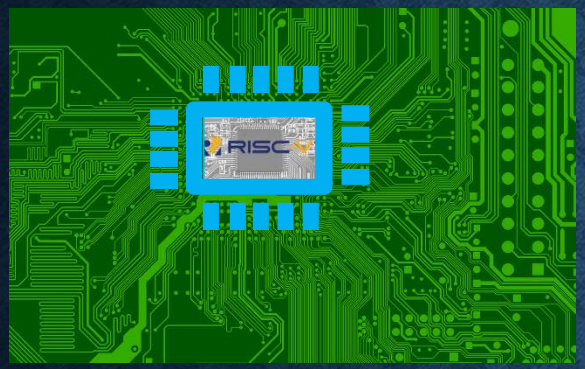
WHAT IS THIS TALK ABOUT?

Hardware:

Software:

WHAT IS THIS TALK ABOUT?

Hardware:



Software:

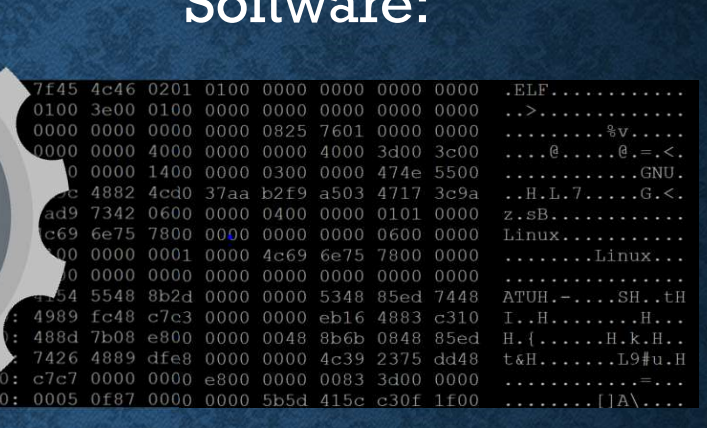
WHAT IS THIS TALK ABOUT?

Hardware:

Software:

WHAT IS THIS TALK ABOUT?

Hardware:



Software:

Pure HW attacks, e.g.:

- Glitching
- Side channel
- Physical probing
- More…

Pure SW attacks, e.g.:

- Memory safety (like overflows)
- Injections (like cmd, XSS, SQL, etc.)
- Logical issues (like bad design)
- More…

WHAT IS THIS TALK ABOUT?

Hardware:

Software:

Pure HW attacks, e.g.:

- Glitching
- Side channel
- Physical probing
- More…

Pure SW attacks, e.g.,:

- Memory safety (like overflows)
- Injection (like command, XSS, SQL, etc.)
- Logical issues (like bad design)
- More…

Targeting specific implementation (e.g., programming language, compiler, firmware, etc.)

WHAT IS THIS TALK ABOUT?

Hardware:

Software:

Pure HW attacks, e.g.,:

- Glitching
- Side channel
- Physical probing
- More…

Targeting specific implementation (e.g., CPU family, implementation of architecture, etc.)

Pure SW attacks, e.g.,:

- Memory safety (like overflows)
- Injection (like XSS, SQL, etc.)
- Logical issues (like bad design)
- More…

Targeting specific implementation (e.g., programming language, compiler, firmware, etc.)

WHAT IS THIS TALK ABOUT?

Hardware:

Software:

Pure HW attacks, e.g.,:

- Glitching
- Side channel
- Physical probing
- More…

Targeting specific implementation (e.g., CPU family, implementation of architecture, etc.)

Pure SW attacks, e.g.,:

- Memory safety (like overflows)
- Injection (like XSS, SQL, etc.)
- Logical issues (like bad design)
- More…

Targeting specific implementation (e.g., programming language, compiler, firmware, etc.)

Mix of HW and SW attacks e.g.:
Spectre / Meltdown

WHAT IS THIS TALK ABOUT?

Hardware:

Software:

Hacker

What if the bug is in the „reference code" like HW ISA itself?

Pure SW attacks, e.g.,:

- Memory safety (like overflows)
- Injection (like XSS, SQL, etc.)
- Logical issues (like bad design)
- More...

Targeting specific implementation (e.g., programming language, compiler, firmware, etc.)

Mix of HW and SW attacks e.g.:
Spectre / Meltdown

WHAT IS THIS TALK ABOUT?

Hardware:

Software:

Pure HW attacks, e.g.,:

- Targeting specific implementation (e.g., CPU family, implementatino of architecture, etc.)
- Physical probing
- More...

What if the bug is in the „reference code" like HW ISA itself?

Pure SW attacks, e.g.,:

- Memory safety (like overflows)
- Injection (like XSS, SQL, etc.)
- Logical issues (like bad design)
- More...

Targeting specific implementation (e.g., programming language, compiler, firmware, etc.)

- Problem with **all implementations** not a specific one!
- SW can't trust HW at all…

Mix of HW and SW attacks e.g.: Spectre / Meltdown

WHAT IS THIS TALK ABOUT?

Hardware:

Software:

What if the bug is in the „reference code" like HW ISA itself?

Targeting specific implementation (e.g., programming language, compiler, firmware, etc.)

Mix of HW and SW attacks e.g.: Spectre / Meltdown

- Problem with **all implementations** not a specific one!
- SW can't trust HW at all…

HOW DID WE FIND IT?

- We wanted to analyze Boot-SW where specific microcode runs but…
  - It was running on the RISC-V chip (which we had 0 experience with)
    - Moreover, it was a custom implementation of RISC-V with custom extensions and functionalities!
  - Boot-SW was written in AdaCore/SPARK language (which we had 0 experience with):
    - Is there any public offensive research on that language?
    - Did anyone ever hear about it before?
  - At that time none of the Reverse Engineering tools natively supported RISC-V
    - Including IDA Pro and Ghidra

HOW DID WE FIND IT?

- We wanted to analyze Boot-SW where specific microcode runs but…
  - It was running on the RISC-V chip (which we had 0 experience with)
    - Moreover, it was a custom implementation of RISC-V with custom extensions and functionalities!
  - Boot-SW was written in AdaCore/SPARK language (which we had 0 experience with):
    - Is there any public offensive research on that language?
    - Did anyone ever hear about it before?
  - At that time none of the Reverse Engineering tools natively supported RISC-V
    - Including IDA Pro and Ghidra
  - During this talk we will describe our journey through all of the problems which resulted in a discovery of the ambiguity of the RISC-V specification
    - And one additional problem as well ;-)

RISC-V IN A NUTSHELL

- RISC-V is an open standard instruction set architecture (ISA) based on established RISC principles
- Unlike most other ISAs, the RISC-V ISA is provided under **open-source licenses** that do not require fees to use
  - The same RISC-V chip might have tons of different implementations
- RISC-V has a small standard base ISA, with multiple standard extensions:
  - Potential huge fragmentation of the silicons
- Everyone can easily add their own custom RISC-V extension (it's open source!)
  - Even bigger fragmentation!
- There are more than 500+ members of the RISC-V Foundation

RISC-V IN A NUTSHELL

RISC-V VS X86

	x86(-64)	RISC-V
License	Fees for ISA and microarchitecture	No fee for ISA & microarchitecture
Instruction Set	CISC*	RISC
ISA variants	16 / 32 / 64 bits	32 / 64 / 128 bits
Memory model	Register-memory architecture	Load-store architecture
Registers	16-bit: 6 semi-dedicated registers, BP and SP are not general-purpose 32-bit: 8 GPRs, including EBP and ESP 64-bit: 16 GPRs, including RBP and RSP	32 (16 in the embedded variant) – including one always-zero register
XOM	Only using SLAT – requires hypervisor	Everywhere
SW ecosystem support	Linux, Windows, MacOS, more…	Linux only…

* Since Pentium Pro, x86 instructions are turned into micro ops (kind of like RISC)

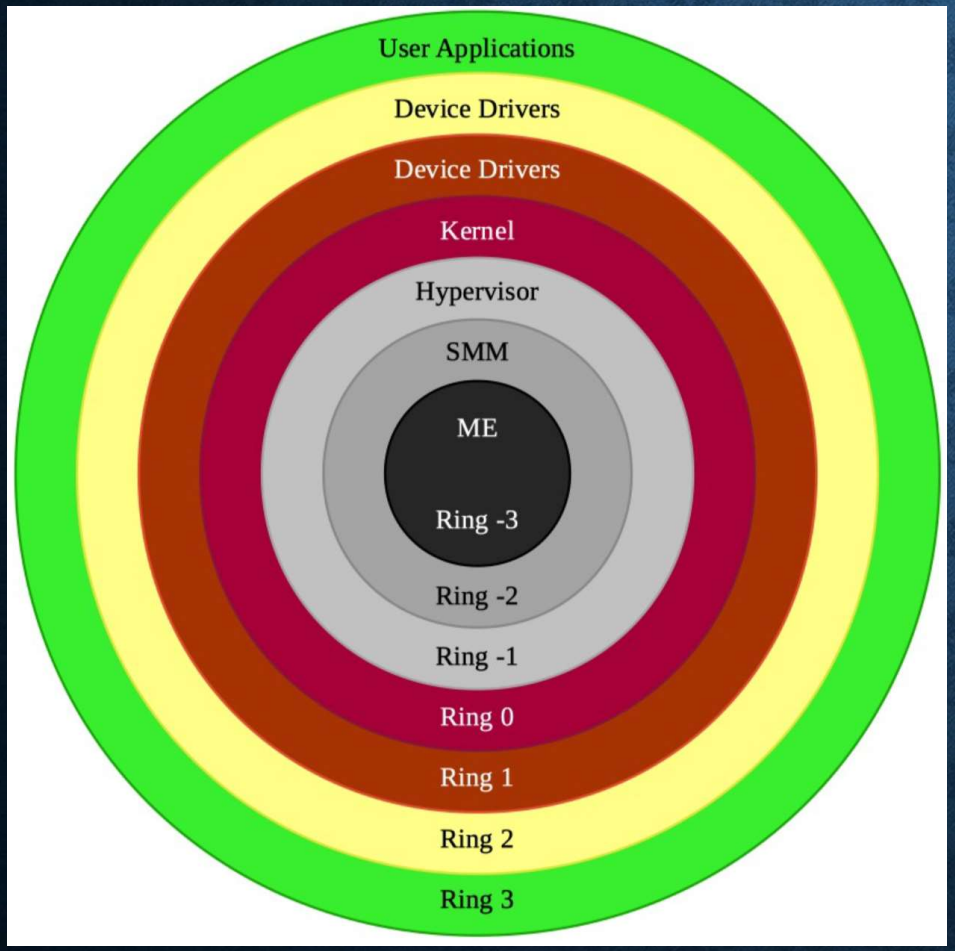
RISC-V VS X86

- Privilege modes / levels

RISC-V VS X86

- Privilege modes / levels

X86(-64):

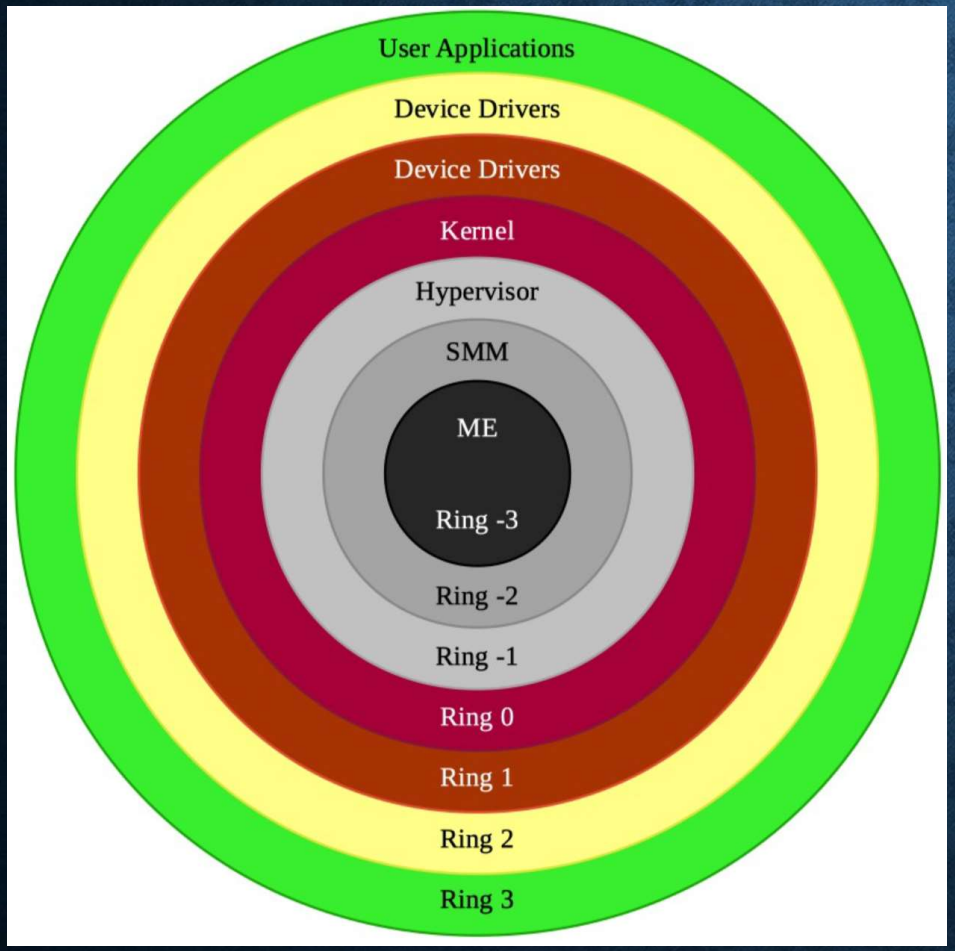


https://medium.com/swlh/negative-rings-in-intel-architecture-the-security-threats-youve-probably-never-heard-of-d725a4b6f831

RISC-V VS X86

- Privilege modes / levels

X86(-64):

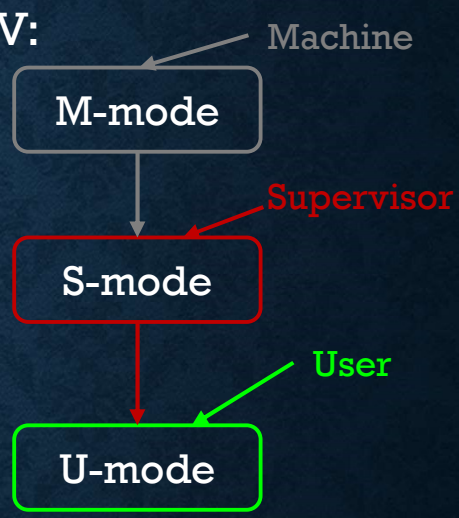


https://medium.com/swlh/negative-rings-in-intel-architecture-the-security-threats-youve-probably-never-heard-of-d725a4b6f831

RISC-V:

Supported combinations:

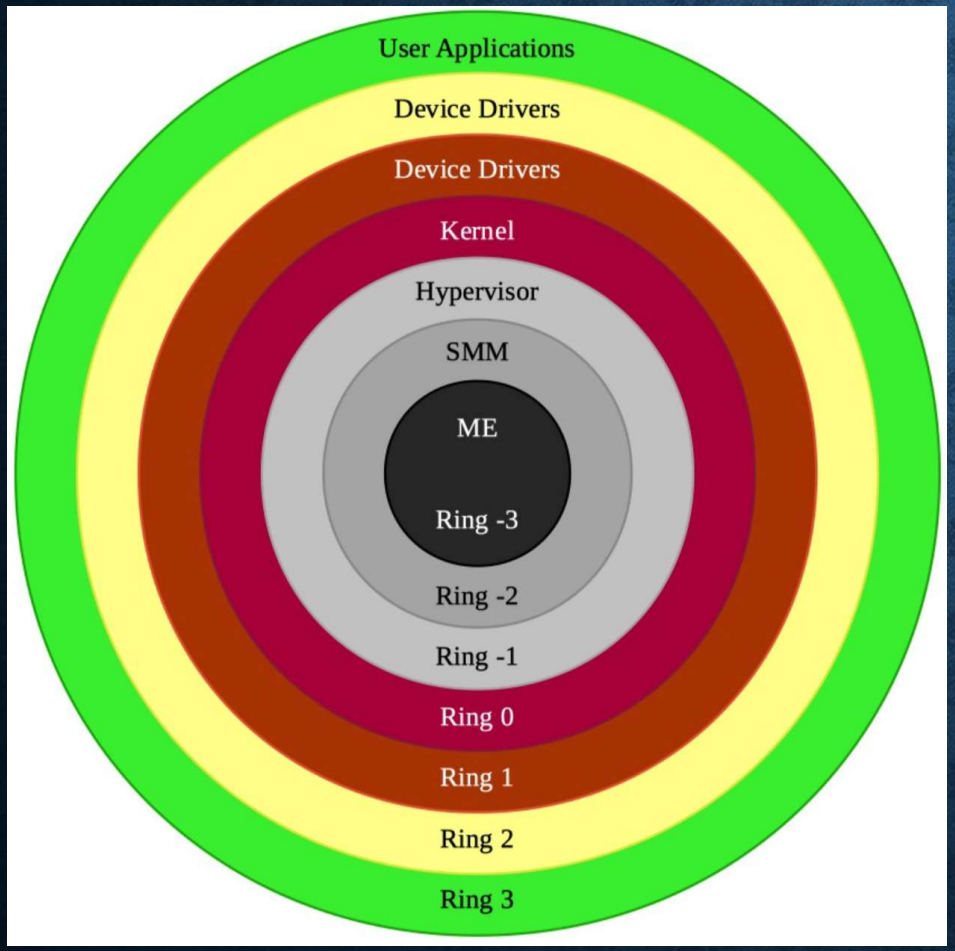
- M
- M + U
- M + S + U



RISC-V VS X86

- Privilege modes / levels

X86(-64):

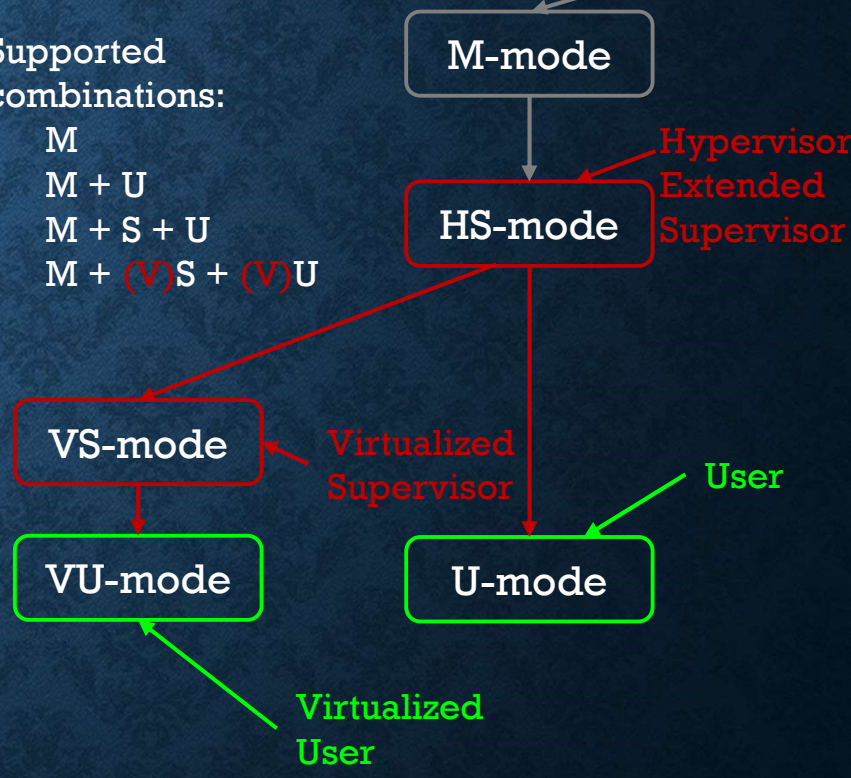


https://medium.com/swlh/negative-rings-in-intel-architecture-the-security-threats-youve-probably-never-heard-of-d725a4b6f831

RISC-V:

Supported combinations:

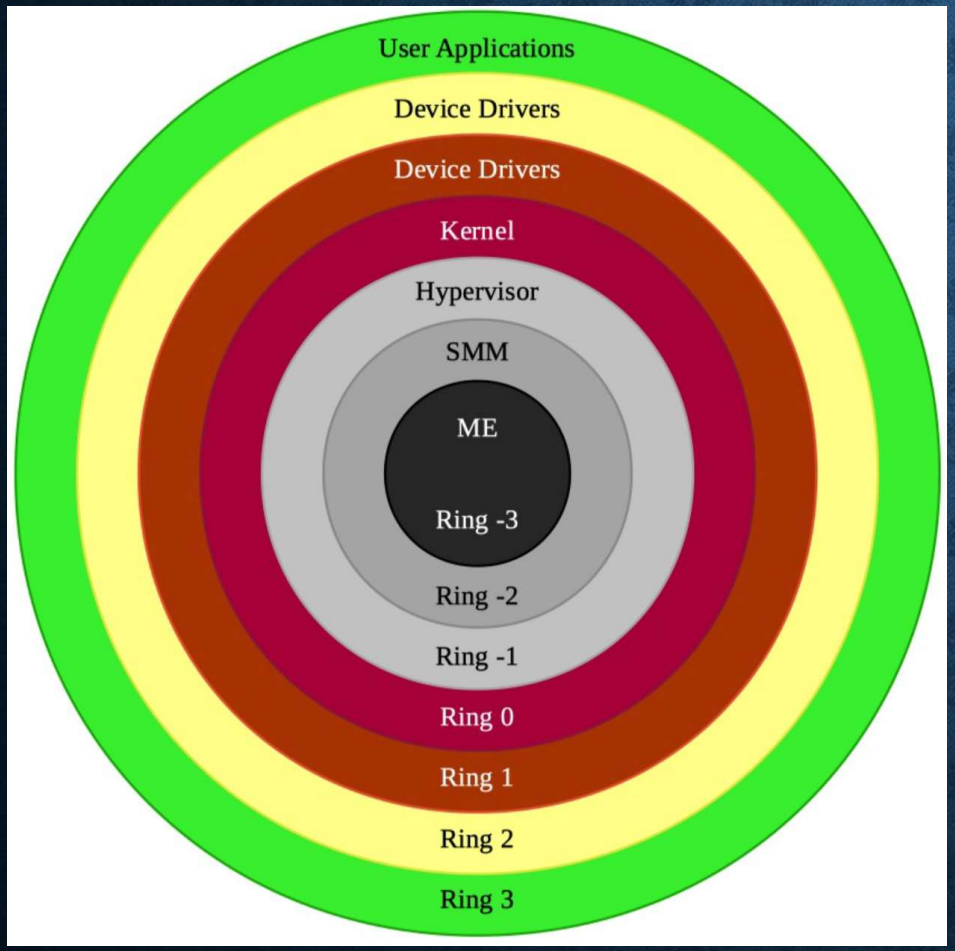
- M
- M + U
- M + S + U
- M + (V)S + (V)U



RISC-V VS X86

❖ Privilege modes / levels

X86(-64):

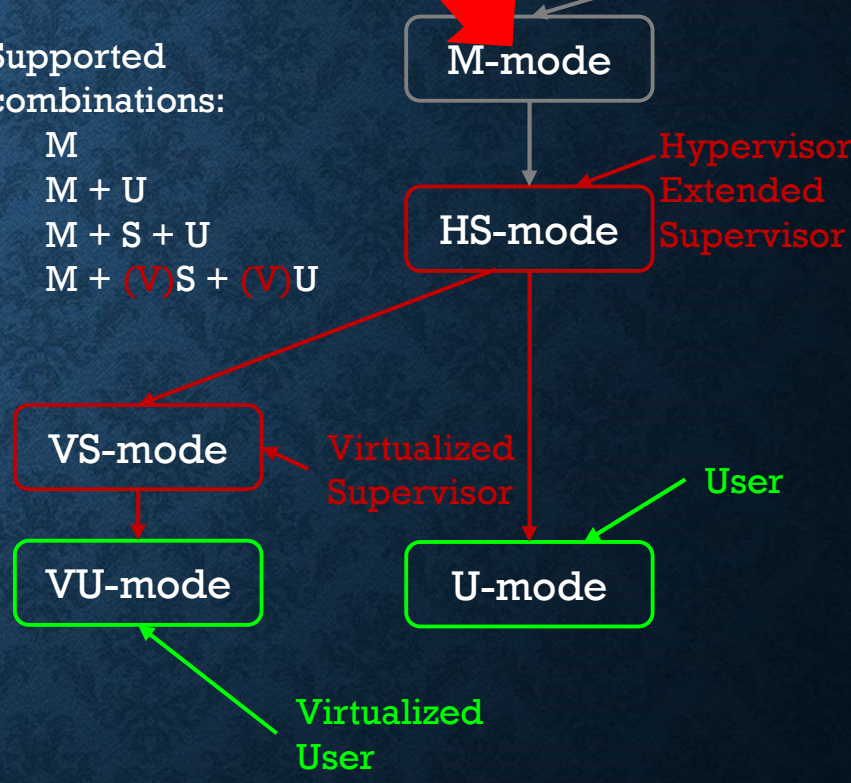


RISC-V:

"GOD" MODE

Supported combinations:

- M
- M + U
- M + S + U
- M + (V)S + (V)U



https://medium.com/swlh/negative-rings-in-intel-architecture-the-security-threats-youve-probably-never-heard-of-d725a4b6f831

ADACORE / SPARK

SPARK 2014

Expanding the boundaries of safe and secure programming.

ADACORE / SPARK

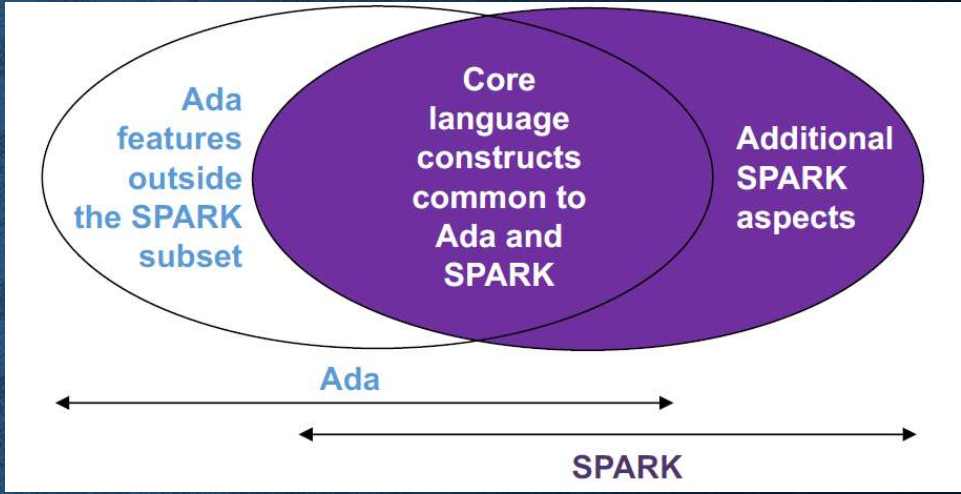
SPARK 2014

Expanding the boundaries of safe and secure programming.

What the…. ?

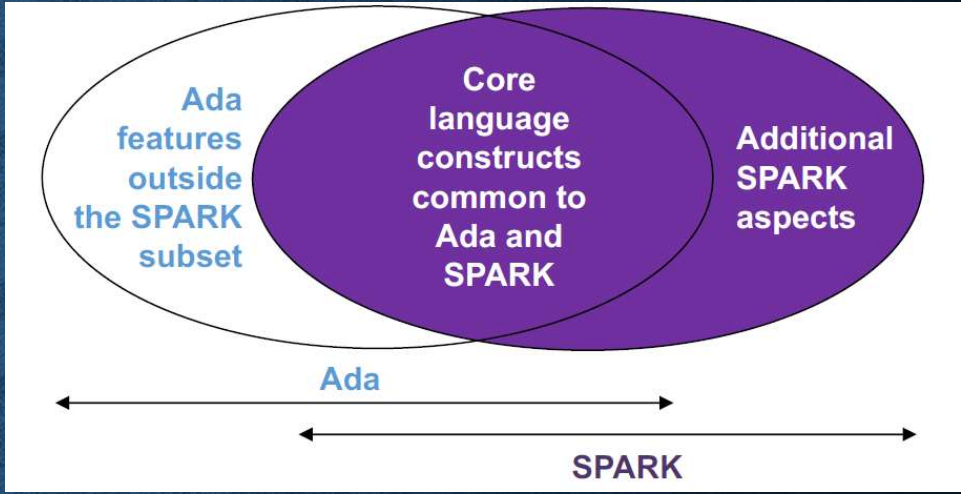
WHAT IS ADACORE/SPARK?

- Programming language + set of analysis tools
  - The strength is in the analysis tools…
  - GNATProve, GNATStack, GNATTest, GNATEmulator



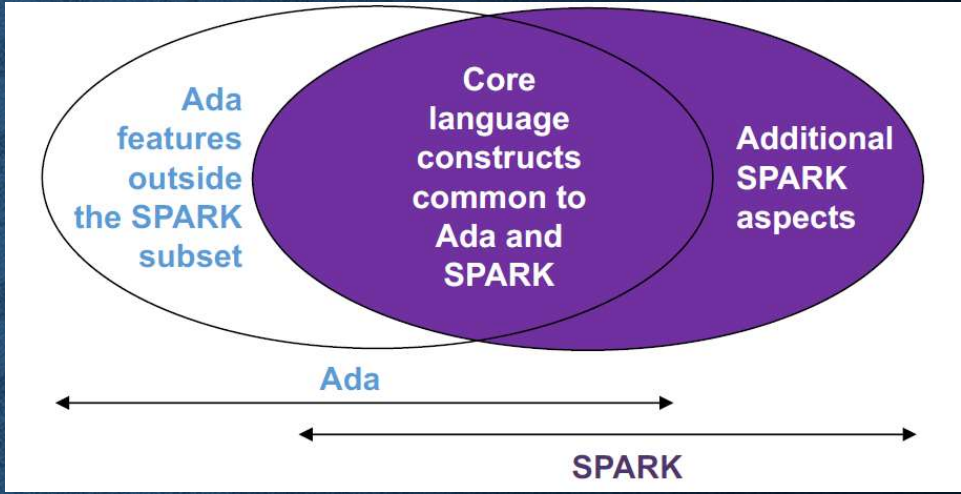
WHAT IS ADACORE/SPARK?

- Programming language + set of analysis tools
  - The strength is in the analysis tools…
  - GNATProve, GNATStack, GNATTest, GNATEmulator
- Statically provable
  - Proves that dynamic checks cannot fail
  - Absence of Run-Time Errors
  - Formal verification (Proofs)



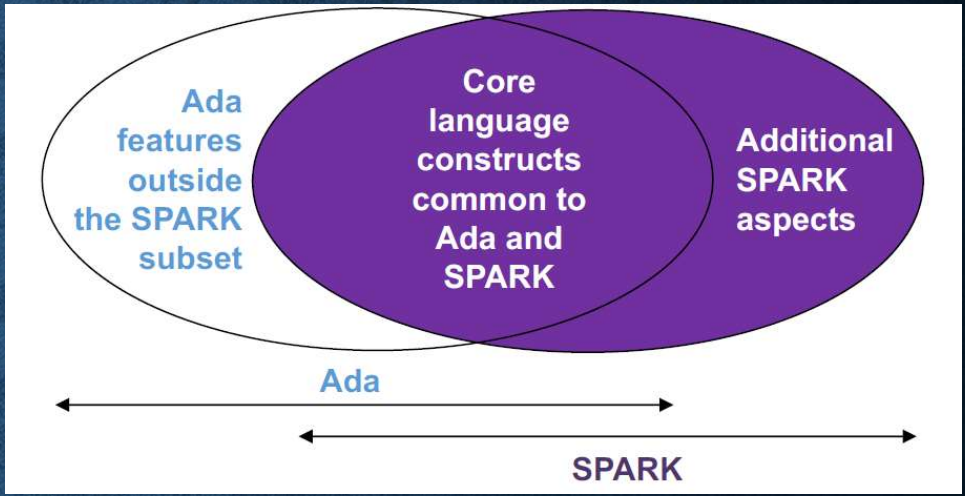
WHAT IS ADACORE/SPARK?

- Programming language + set of analysis tools
  - The strength is in the analysis tools…
  - GNATProve, GNATStack, GNATTest, GNATEmulator
- Statically provable
  - Proves that dynamic checks cannot fail
  - Absence of Run-Time Errors
  - Formal verification (Proofs)
- Memory safe language (like RUST)



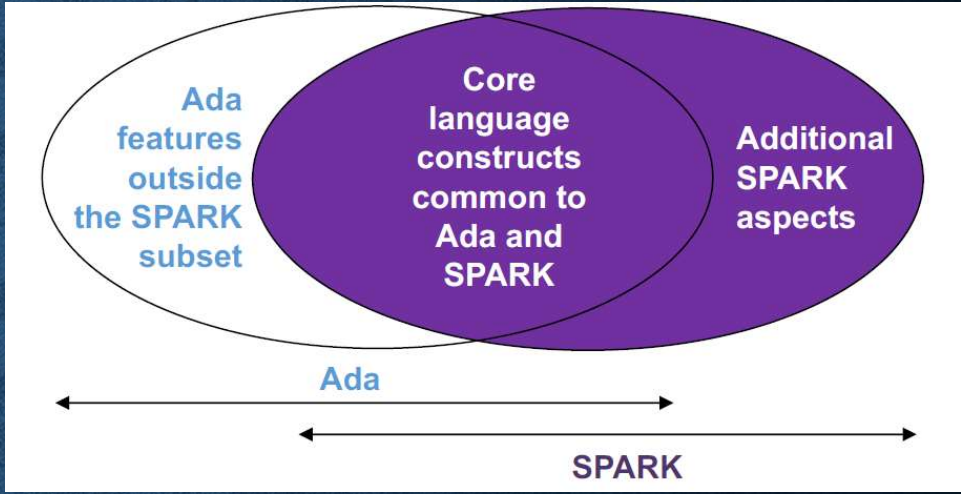
WHAT IS ADACORE/SPARK?

- Programming language + set of analysis tools
  - The strength is in the analysis tools…
  - GNATProve, GNATStack, GNATTest, GNATEmulator
- Statically provable
  - Proves that dynamic checks cannot fail
  - Absence of Run-Time Errors
  - Formal verification (Proofs)
- Memory safe language (like RUST)
- Very strong typing system (much stronger than RUST)
  - No arithmetic overflows, integer overflows, etc.



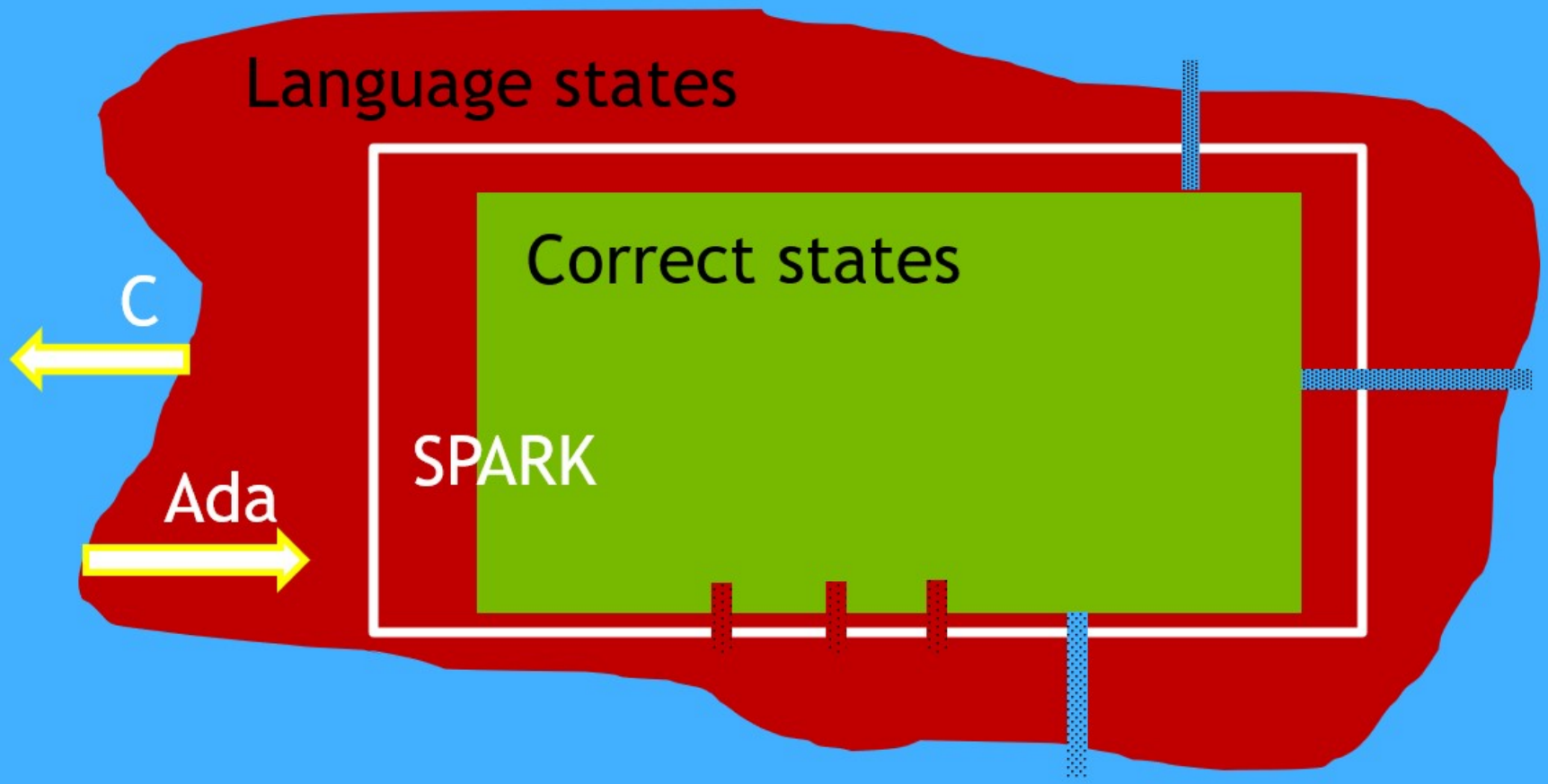
WHAT IS ADACORE/SPARK?

- Programming language + set of analysis tools
  - The strength is in the analysis tools…
  - GNATProve, GNATStack, GNATTest, GNATEmulator
- Statically provable
  - Proves that dynamic checks cannot fail
  - Absence of Run-Time Errors
  - Formal verification (Proofs)
- Memory safe language (like RUST)
- Very strong typing system (much stronger than RUST)
  - No arithmetic overflows, integer overflows, etc.
- Traditionally used in industries such as:
  - Avionics, Railways, Defense, Auto, IoT



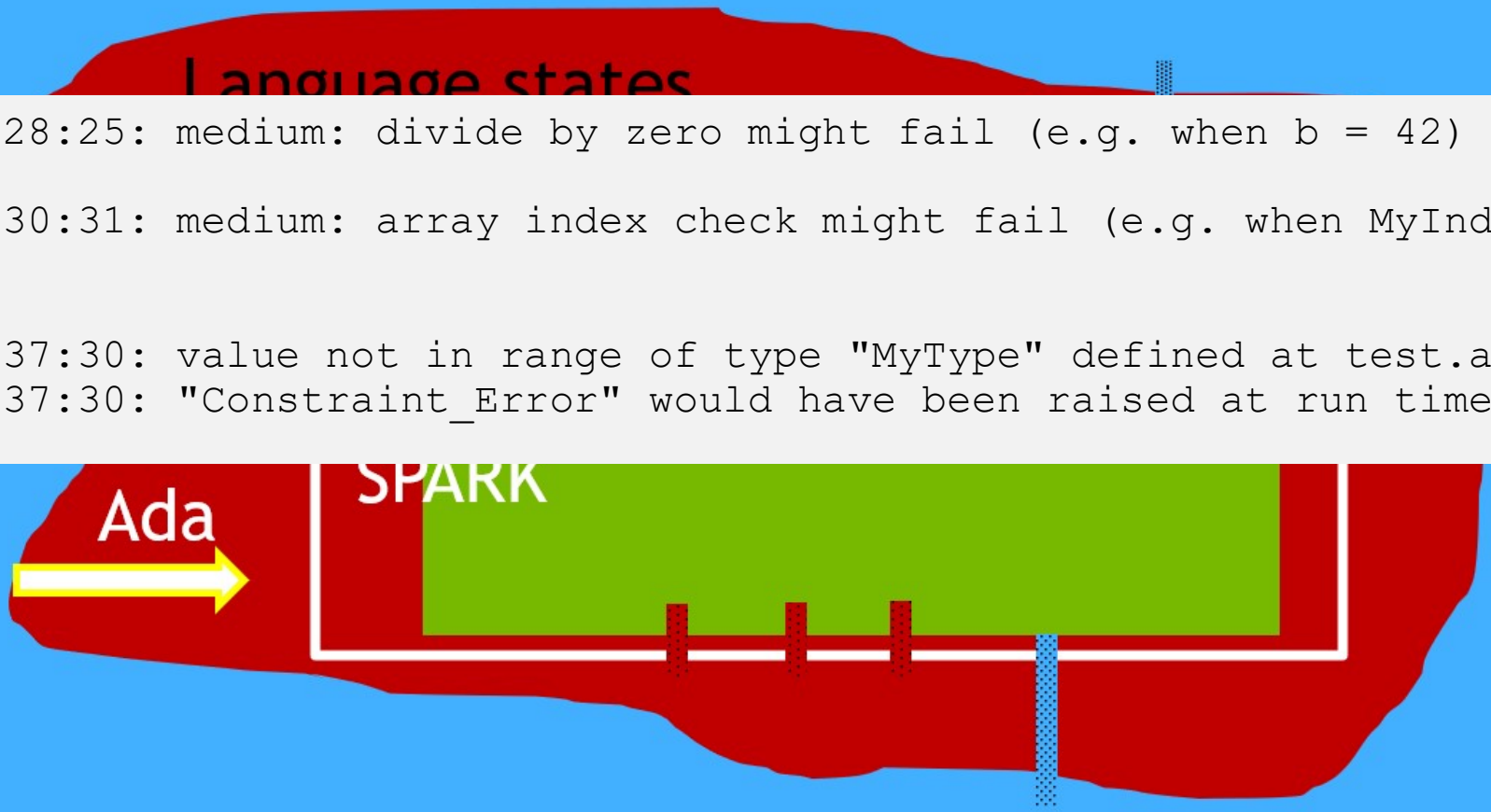
WHAT IS ADACORE/SPARK?

Machine states



WHAT IS ADACORE/SPARK?

Machine states




```
test.adb:28:25: medium: divide by zero might fail (e.g. when b = 42)

test.adb:30:31: medium: array index check might fail (e.g. when MyIndex =
36)

test.adb:37:30: value not in range of type "MyType" defined at test.ads:6
test.adb:37:30: "Constraint_Error" would have been raised at run time
```

Machin

```
test.adb
test.adb
36)
test.adb
test.adb
```

```
C:\GNAT\Tmp\1>gnatprove -P pi3_test -j0 --ide-progress-bar --steps=30000 --prover=all --
assumptions --proof-warnings
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
completed 1 out of 2 (50%)...
pi3_test.adb:18:18: warning: unreachable code[#0]
completed 2 out of 2 (100%)...
Summary logged in C:\GNAT\Tmp\1\gnatprove\gnatprove.out
```

```
C:\GNAT\Tmp\1>gprbuild p_run.adb -cargs -fcallgraph-info=su
using project file pi3_test.gpr
Compile
  [Ada]        p_run.adb
  [Ada]        pi3_test.adb
Bind
  [gprbind]     p_run.bexch
  [Ada]        p_run.ali
Link
  [link]       p_run.adb
```

GNATprove doesn't see any problems with this project

GNATstack detected potential problem where prover didn't

```
C:\GNAT\Tmp\1>gnatstack *.ci
Worst case analysis is *not* accurate because of unbounded frames, external calls. Use -Wa for
details.

Accumulated stack usage information for entry points

main : total 10737872+? bytes
 +-> main
 +-> p_run
 +-> pi3_test.pi3_run *
 +-> <__gnat_rcheck_CE_Index_Check> *

GNATstack: analysis successfully finished
C:\GNAT\Tmp\1>
```

Lessons learned:

- You can compile buggy code – problems are detected by the tools and developers might not run them at all!
- Tools are orthogonal and detect different classes of problems – to be fully protected you must run all of them!
- What are the classes of problems which can or cannot be detected? – very limited public information :(

Lessons learned:

- You can compile buggy code – problems are detected by the tools and developers might not run them at all!
- Tools are orthogonal and detect different classes of problems – to be fully protected you must run all of them!
- What are the classes of problems which can or cannot be detected? – very limited public information :(- time for more research!

ADACORE/SPARK - EVALUATION

ADACORE/SPARK - EVALUATION

General memory corruption			
Type of Problem \ Language	C	C++	SPARK
Classic buffer overflow (heap / stack / .bss / more)	Vulnerable	Might be limited in new standard but still possible	Safe
Buffer underflow	Vulnerable	Might be limited in new standard but still possible	Safe
Out-of-bound read / write	Vulnerable	Might be limited in new standard but still possible	Safe
Improper Validation of Array Index	Vulnerable	Might be limited in new standard but still possible	Safe
Off-by-one (over/under flow of an allocated buffer)	Vulnerable	Might be limited in new standard but still possible	Safe
Incorrect Calculation of Buffer Size	Vulnerable	Vulnerable	Safe
Reliance on Data/Memory Layout / Padding	Vulnerable	Vulnerable	Safe
Use of Inherently or Potentially Dangerous Function	Vulnerable	Might be limited in new standard but still possible	Safe
Improper Clearing of Heap Memory Before Release	Vulnerable	Vulnerable	Safe
Double Free	Vulnerable	Might be limited in new standard but still possible	Safe
Use After Free	Vulnerable	Vulnerable	Safe**
Use of Uninitialized Variable	Vulnerable	Might be limited in new standard but still possible	Safe
Memory Leak	Vulnerable	Vulnerable	Safe*

**if a developer mixes SPARK with other programming languages (e.g. ADA) where function/procedure has a SPARK spec and body not in SPARK, prover might make a presumption that user ensures certain validation. However, user might make a mistake and if ADA pointers (access type) was used, it is possible to leak dynamically allocated memory.*

***if access type is freed, ADA zeros it. If user is not aware that memory was freed. it will always read zero as a value. However, there might be a case that behavior/flow of the program depends on that value.*

ADACORE/SPARK - EVALUATION

General memory corruption			
Type of Problem \ Language	C	C++	SPARK
Classic buffer overflow (heap / stack / .bss / more)	Vulnerable	Might be limited in new standard but still possible	Safe
Buffer underflow	Vulnerable	Might be limited in new standard but still possible	Safe
Out-of-bound read / write	Vulnerable	Might be limited in new standard but still possible	Safe
Improper Validation of Array Index	Vulnerable	Might be limited in new standard but still possible	Safe
Off-by-one (over/under flow of an allocated buffer)	Vulnerable	Might be limited in new standard but still possible	Safe
Incorrect Calculation of Buffer Size	Vulnerable	Vulnerable	Safe
Reliance on Data/Memory Layout / Padding	Vulnerable	Vulnerable	Safe
Use of Inherently or Potentially Dangerous Function	Vulnerable	Might be limited in new standard but still possible	Safe
Improper Clearing of Heap Memory Before Release	Vulnerable	Vulnerable	Safe
Double Free	Vulnerable	Might be limited in new standard but still possible	Safe
Use After Free	Vulnerable	Vulnerable	Safe**
Use of Uninitialized Variable	Vulnerable	Might be limited in new standard but still possible	Safe
Memory Leak	Vulnerable	Vulnerable	Safe*

**if a developer mixes SPARK with other programming languages (e.g. ADA) where function/procedure has a SPARK spec and body not in SPARK, prover might make a presumption that user ensures certain validation. However, user might make a mistake and if ADA pointers (access type) was used, it is possible to leak dynamically allocated memory.*
***if access type is freed, ADA zeros it. If user is not aware that memory was freed. it will always read zero as a value. However, there might be a case that behavior/flow of the program depends on that value.*

General pointers' security			
Type of Problem \ Language	C	C++	SPARK
Improper Null Termination	Vulnerable	Might be limited in new standard but still possible	N/A
NULL Pointer Dereference	Vulnerable	Might be safe (references)	Safe
Use of sizeof() on a Pointer Type	Vulnerable	Vulnerable	N/A
Incorrect Pointer Scaling	Vulnerable	Might be safe (smart pointers) in new standard but still possible	N/A
Use of Pointer Subtraction to Determine Size	Vulnerable	Might be limited in new standard but still possible	N/A
Assignment of a Fixed Address to a Pointer	Vulnerable	Vulnerable	N/A
Uncontrolled Memory Allocation	Vulnerable	Vulnerable	Vulnerable*
Return of Stack Variable Address	Vulnerable	Vulnerable	N/A
Dangling Pointers	Vulnerable	Vulnerable	N/A**
Type confusion	Vulnerable	Vulnerable	Might be possible if mixed with non-SPARK code
Double Fetch	Vulnerable	Vulnerable	Might be possible

**if a developer mixes SPARK with other programming languages (e.g. ADA) where function/procedure has a SPARK spec and body not in SPARK, prover might make a presumption that user ensures certain validation. However, user might make a mistake and if data type has discriminants (<>) depending on non-SPARK values (e.g. ADA types), uncontrolled memory allocation is possible. Similar problem might exist during uncontrolled call graph flow which can dynamically pressure the stack. Nevertheless, these problems might be detected by* GNATstack *tool and it is very important to not rely only on the prover.*
***It is the same situation as described in "General memory corruption" point *. Not freed access type might generate a problematic variant known as "dangling references". This is only possible in a non-SPARK part of the code (with SPARK spec) through incorrect uses of* Unchecked_Deallocation[4]

ADACORE/SPARK - EVALUATION

Arithmetic security			
Type of Problem \ Language	C	C++	SPARK
Integer Underflow	Vulnerable	Might be limited (SafeInt) but in general still possible	Safe
Integer Overflow	Vulnerable	Might be limited (SafeInt) but in general still possible	Safe
Arithmetic Overflow	Vulnerable	Might be limited (SafeInt) but in general still possible	Safe
Numeric Truncation Error	Vulnerable	Vulnerable	Safe
Signed / unsigned conversion error	Vulnerable	Vulnerable	Safe
Divide by zero	Vulnerable	Vulnerable	Safe

ADACORE/SPARK - EVALUATION

Arithmetic security			
Type of Problem \ Language	C	C++	SPARK
Integer Underflow	Vulnerable	Might be limited (SafeInt) but in general still possible	Safe
Integer Overflow	Vulnerable	Might be limited (SafeInt) but in general still possible	Safe
Arithmetic Overflow	Vulnerable	Might be limited (SafeInt) but in general still possible	Safe
Numeric Truncation Error	Vulnerable	Vulnerable	Safe
Signed / unsigned conversion error	Vulnerable	Vulnerable	Safe
Divide by zero	Vulnerable	Vulnerable	Safe

Misc / other			
Type of Problem \ Language	C	C++	SPARK
Use of Externally-Controlled Format String	Vulnerable	Might be limited but in general still possible	Safe
Missing Default Case in Switch Statement	Vulnerable	Vulnerable	Safe
Assigning instead of Comparing and Otherwise	Vulnerable	Vulnerable	Safe
Function Call with Incorrect Arguments	Vulnerable	Vulnerable	Safe

ADACORE/SPARK - EVALUATION

Arithmetic security			
Type of Problem \ Language	C	C++	SPARK
Integer Underflow	Vulnerable	Might be limited (SafeInt) but in general still possible	Safe
Integer Overflow	Vulnerable	Might be limited (SafeInt) but in general still possible	Safe
Arithmetic Overflow	Vulnerable	Might be limited (SafeInt) but in general still possible	Safe
Numeric Truncation Error	Vulnerable	Vulnerable	Safe
Signed / unsigned conversion error	Vulnerable	Vulnerable	Safe
Divide by zero	Vulnerable	Vulnerable	Safe

Misc / other			
Type of Problem \ Language	C	C++	SPARK
Use of Externally-Controlled Format String	Vulnerable	Might be limited but in general still possible	Safe
Missing Default Case in Switch Statement	Vulnerable	Vulnerable	Safe
Assigning instead of Comparing and Otherwise	Vulnerable	Vulnerable	Safe
Function Call with Incorrect Arguments	Vulnerable	Vulnerable	Safe

Parallel execution security			
Type of Problem \ Language	C	C++	SPARK
Race condition	Vulnerable	Vulnerable	Might be limited*
Signal Handler Race Condition	Vulnerable	Vulnerable	TBD
Unsafe Function Call from a Signal Handler	Vulnerable	Vulnerable	Might be possible**
Race Condition in Switch() Statement	Vulnerable	Vulnerable	Might be limited*
Deadlock	Vulnerable	Vulnerable	Might be limited*
Passing Mutable Objects to an Untrusted Method	Vulnerable	Vulnerable	Might be possible**
Improper Cleanup on Thrown Exception	Vulnerable	Vulnerable	Vulnerable***

**Might be limited by "protected objects" and appropriate modeled in Ravenscar*
***Might be possible if function call (or method) is coming to the language which is not trusted/secured*
****In generic case SPARK won't be able to help unless developer write specific contracts that reflected requirements (in this case "cleanup" requirement). But if the requirement was indeed modeled, then SPARK prover will catch an implementation mistake*

ADACORE/SPARK - EVALUATION

Arithmetic security			
Type of Problem / Language	C	C++	SPARK
Integer Underflow	Vulnerable	Might be limited (SafeInt) but in general still possible	Safe
Integer Overflow	Vulnerable	Might be limited (SafeInt) but in general still possible	Safe
Arithmetic Overflow	Vulnerable	Might be limited (SafeInt) but in general still possible	Safe
Numeric Truncation Error	Vulnerable	Vulnerable	Safe
Signed / unsigned conversion error	Vulnerable	Vulnerable	Safe
Divide by zero	Vulnerable	Vulnerable	Safe

Misc / other			
Type of Problem / Language	C	C++	SPARK
Use of Externally-Controlled Format String	Vulnerable	Might be limited but in general still possible	Safe
Missing Default Case in Switch Statement	Vulnerable	Vulnerable	Safe
Assigning instead of Comparing and Otherwise	Vulnerable	Vulnerable	Safe
Function Call with Incorrect Arguments	Vulnerable	Vulnerable	Safe

Parallel execution security			
Type of Problem / Language	C	C++	SPARK
Race condition	Vulnerable	Vulnerable	Might be limited*
Signal Handler Race Condition	Vulnerable	Vulnerable	TBD
Unsafe Function Call from a Signal Handler	Vulnerable	Vulnerable	Might be possible**
Race Condition in Switch() Statement	Vulnerable	Vulnerable	Might be limited*
Deadlock	Vulnerable	Vulnerable	Might be limited*
Passing Mutable Objects to an Untrusted Method	Vulnerable	Vulnerable	Might be possible**
Improper Cleanup on Thrown Exception	Vulnerable	Vulnerable	Vulnerable***

**Might be limited by "protected objects" and appropriate modeled in Ravenscar*
***Might be possible if function call (or method) is coming to the language which is not trusted/secured*
****In generic case SPARK won't be able to help unless developer write specific contracts that reflected requirements (in this case "cleanup" requirement). But if the requirement was indeed modeled, then SPARK prover will catch an implementation mistake*

Logic bugs			
Type of Problem / Language	C	C++	SPARK
General logic error	Vulnerable	Vulnerable	Vulnerable
Bad design	Vulnerable	Vulnerable	Vulnerable
Inaccurate modeling of hardware	Vulnerable	Vulnerable	Vulnerable
Inaccurate handling of DMA*	Vulnerable	Vulnerable	Vulnerable
Rely on the behavior from the non-SPARK code which can be badly design / implemented*	Vulnerable	Vulnerable	Vulnerable
Aliasing with overlays*	Vulnerable	Vulnerable	Vulnerable
Confidential / privacy data leak*	Vulnerable	Vulnerable	Vulnerable
Multiple threads stack collision*	Vulnerable	Vulnerable	Vulnerable

ADACORE/SPARK - EVALUATION

- Lesson learned:
  - You can compile buggy code – problems are detected by the tools and developers might not run them at all!
  - Tools are orthogonal and detect different classes of problems – to be fully protected you must run all of them!

ADACORE/SPARK - EVALUATION

- Lesson learned:
  - You can compile buggy code – problems are detected by the tools and developers might not run them at all!
  - Tools are orthogonal and detect different classes of problems – to be fully protected you must run all of them!
  - Most of the potential security issues might be:
    - In the design
    - Logical errors

ADACORE/SPARK - EVALUATION

- Lesson learned:
  - You can compile buggy code – problems are detected by the tools and developers might not run them at all!
  - Tools are orthogonal and detect different classes of problems – to be fully protected you must run all of them!
  - Most of the potential security issues might be:
    - In the design
    - Logical errors
  - Bugs can be introduced by the compiler itself as well

ADACORE/SPARK - EVALUATION

- Lesson learned:
  - You can compile buggy code – problems are detected by the tools and developers might not run them at all!
  - Tools are orthogonal and detect different classes of prob s – to be fully protected you must run all of them!
  - Most of the potential security issues might be:
    - In the design
    - Logical errors
  - Bugs can be introduced by the compiler itself as well

We need to analyze the binary!

ADACORE/SPARK - EVALUATION

- Lesson learned:
  - You can compile buggy code – problems are detected by the tools and developers might not run them at all!
  - Tools are orthogonal and detect different classes of prob s – to be fully protected you must run all of them!
  - Most of the potential security issues might be:
    - In the design
    - Logical errors
  - Bugs can be introduced by the compiler itself as well

We need to analyze the binary!

During this research, neither IDA Pro nor Ghidra supported RISC-V ; (

ADACORE/SPARK - EVALUATION

- Lesson learned:
  - You can compile buggy code – problems are detected by the tools and developers might not run them at all!
  - Tools are orthogonal and detect different classes of prob s – to be fully protected you must run all of them!
  - Most of the potential security issues might be:
    - In the design
    - Logical errors
  - Bugs can be introduced by the compiler itself as well

We need to analyze the binary!

During this research, neither IDA Pro nor Ghidra supported RISC-V ; (

BRINGING RISC-V TO GHIDRA

- Ghidra 9.0 didn't support RISC-V…
  - Moreover, we were dealing with the custom RISC-V with the custom extensions…

BRINGING RISC-V TO GHIDRA

- Ghidra 9.0 didn't support RISC-V…
  - Moreover, we were dealing with the custom RISC-V with the custom extensions…
- RISC-V is huge!
  - Implementing entire RISC-V base would take TONS of time…
  - … additionally, we needed custom RISC-V extension support

BRINGING RISC-V TO GHIDRA

- Ghidra 9.0 didn't support RISC-V…
  - Moreover, we were dealing with the custom RISC-V with the custom extensions…
- RISC-V is huge!
  - Implementing entire RISC-V base would take TONS of time…
  - … additionally, we needed custom RISC-V extension support
- We found on the github a few RISC-V base plugins – different implementations:
  - We decided to "integrate" one of the plugin to Ghidra TOT
  - … Few months after our research Ghidra 9.2 brought RISC-V support using exactly the same plugin ;-)

BRINGING RISC-V TO GHIDRA

- Where to start?
  - We successfully integrated RISC-V plugin, but we needed to modify it…
- Ghidra is using SLEIGH language to describe the CPU
  - SLEIGH is a processor specification language developed for Ghidra (heritage from the SLED)
  - Very little documentation about it
  - If you want to model a simple CPU, it's fine, but a more complex one could be very painful (at least it was for me ;-))
  - We used already supported CPUs as a "source of knowledge"
  - Additionally, we found only one really useful resource - Guillaume Valadon presentation: https://guedou.github.io/talks/2019_BeeRump/slides.pdf
- You need to create a cspec, ldefs, pspec, slaspec, and a Module.manifest file:
  - We already had it, but we needed to modify slaspec
    - You define there the register definitions, aliases, instructions etc.
  - Ghidra can be compiled with a bad SLASPEC if its syntax is correct:
    - Then you will see on runtime if it works, or you will see tons of JAVA exceptions
    - We used "check & try" + "calm down" technique to achieve what we wanted :)

BRINGING RISC-V TO GHIDRA

```
define token instr (32)
  op0001=(0,1)
  op0204=(2,4)
  op0506=(5,6)
  op0707=(7,7)
  op0711=(7,11)
  r0711=(7,11)
  fr0711=(7,11)
  op0811=(8,11)
  op1214=(12,14)
  funct3=(12,14)
  op1219=(12,19)
  op1231=(12,31)
  sop1231=(12,31) signed
  op1519=(15,19)
  r1519=(15,19)
  fr1519=(15,19)
  op1527=(15,27)
  op1531=(15,31)
  op2020=(20,20)
  succ=(20,23)
  op2024=(20,24)
  r2024=(20,24)
  fr2024=(20,24)
  csr_0=(20,27)
```

rt?

sfully integrated RISC-V plugin, but we needed to modify it…

ng SLEIGH language to describe the CPU

a processor specification language developed for Ghidra (heritage from the

documentation about it

to model a simple CPU, it's fine, but a more complex one could be very painful

was for me ;-))

ready supported CPUs as a "source of knowledge"

ly, we found only one really useful resource - Guillaume Valadon presentation:

edou.github.io/talks/2019_BeeRump/slides.pdf

- You need to create a cspec, ldefs, pspec, slaspec, and a Module.manifest file:
  - We already had it, but we needed to modify slaspec
    - You define there the register definitions, aliases, instructions etc.
  - Ghidra can be compiled with a bad SLASPEC if its syntax is correct:
    - Then you will see on runtime if it works, or you will see tons of JAVA exceptions
    - We used "check & try" + "calm down" technique to achieve what we wanted :)

BRINGING RISC-V TO GHIDRA

```
define token instr (32)
  op0001=(0,1)
  op0204=(2,4)
  op0506=(5,6)
  op0707=(7,7)
  op0711=(7,11)
  r0711=(7,11)
  fr0711=(7,11)
  op0811=(8,11)
  op1214=(12,14)
  funct3=(12,14)
  op1219=(12,19)
  op1231=(12,31)
  sop1231=(12,31) signed
  op1519=(15,19)
  r1519=(15,19)
  fr1519=(15,19)
  op1527=(15,27)
  op1531=(15,31)
  op2020=(20,20)
  succ=(20,23)
  op2024=(20,24)
  r2024=(20,24)
  fr2024=(20,24)
  csr_0=(20,27)
```

```
define register offset=0x90000000 size=$(XLEN) [ ustatus ];
define register offset=0x90000010 size=$(XLEN) [ fflags ];
define register offset=0x90000020 size=$(XLEN) [ frm ];
define register offset=0x90000030 size=$(XLEN) [ fcsr ];
```

rt?

sfully inte

ng SLEIGH language to describe the CPU

a processor specification language developed for Ghidra (heritage from the

documentation about it

to model a simple CPU, it's fine, but a more complex one could be very painful

was for me ;-))

ready supported CPUs as a "source of knowledge"

ly, we found only one really useful resource - Guillaume Valadon presentation:

edou.github.io/talks/2019_BeeRump/slides.pdf

- You need to create a cspec, ldefs, pspec, slaspec, and a Module.manifest file:
  - We already had it, but we needed to modify slaspec
    - You define there the register definitions, aliases, instructions etc.
  - Ghidra can be compiled with a bad SLASPEC if its syntax is correct:
    - Then you will see on runtime if it works, or you will see tons of JAVA exceptions
    - We used "check & try" + "calm down" technique to achieve what we wanted :)

BRINGING RISC-V TO GHIDRA

```
define token instr (32)
  op0001=(0,1)
  op0204=(2,4)
  op0506=(5,6)
  op0707=(7,7)
  op0711=(7,11)
  r0711=(7,11)
  fr0711=(7,11)
  op0811=(8,11)
  op1214=(12,14)
  funct3=(12,14)
  op1219=(12,19)
  op1231=(12,31)
  sop1231=(12,31) signed
  op1519=(15,19)
  r1519=(15,19)
  fr1519=(15,19)
  op1527=(15,27)
  op1531=(15,31)
  op2020=(20,20)
  succ=(20,23)
  op2024=(20,24)
  r2024=(20,24)
  fr2024=(20,24)
  csr_0=(20,27)
```

```
define register offset=0x90000000 size=$(XLEN) [ ustatus ];
define register offset=0x90000010 size=$(XLEN) [ fflags ];
define register offset=0x90000020 size=$(XLEN) [ frm ];
define register offset=0x90000030 size=$(XLEN) [ fcsr ];
```

rt?

sfully inte

ng SLEIGH language to describe the CPU

```
attach variables [ csr_0 ]
        [ ustatus fflags frm fcsr uie utvec _ _ _ _ _ _ _ _ _ _
      _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
      _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
      _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
      uscratch uepc ucause utval uip _ _ _ _ _ _ _ _ _ _ _
      _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

a pro

he

docu

to m

ainful

was f

ready supported CPUs as a "source of knowledge"

ly, we found only one really useful resource - Guillaume Valadon presentation:

edou.github.io/talks/2019_BeeRump/slides.pdf

- You need to create a cspec, ldefs, pspec, slaspec, and a Module.manifest file:
  - We already had it, but we needed to modify slaspec
    - You define there the register definitions, aliases, instructions etc.
  - Ghidra can be compiled with a bad SLASPEC if its syntax is correct:
    - Then you will see on runtime if it works, or you will see tons of JAVA exceptions
    - We used "check & try" + "calm down" technique to achieve what we wanted :)

BRINGING RISC-V TO GHIDRA

```
define token instr (32)
  op0001=(0,1)
  op0204=(2,4)
  op0506=(5,6)
  op0707=(7,7)
  op0711=(7,11)
  r0711=(7,11)
  fr0711=(7,11)
  op0811=(8,11)
  op1214=(12,14)
  funct3=(12,14)
  op1219=(12,19)
  op1231=(12,31)
  sop1231=(12,31) signed
  op1519=(15,19)
  r1519=(15,19)
  fr1519=(15,19)
  op1527=(15,27)
  op1531=(15,31)
  op2020=(20,20)
  succ=(20,23)
  op2024=(20,24)
  r2024=(20,24)
  fr2024=(20,24)
  csr_0=(20,27)
```

rt?

sfully inte

```
define register offset=0x90000000 size=$(XLEN) [ ustatus ];
define register offset=0x90000010 size=$(XLEN) [ fflags ];
define register offset=0x90000020 size=$(XLEN) [ frm ];
define register offset=0x90000030 size=$(XLEN) [ fcsr ];
```

ng SLEIGH language to describe the CPU

a pro

```
attach variables [ csr_0 ]
          [ ustatus fflags frm fcsr uie utvec _ _ _ _ _ _ _ _ _ _
        _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
        _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
        _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
        uscratch uepc ucause utval uip _ _ _ _ _ _ _ _ _ _ _
        _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

he

docu

to m

ainful

was f

ready supported CPUs as a "source of knowledge"

ly, we found only one really useful resource - Guillaume Valadon presentation:

edou.github.io/talks/2019_BeeRump/slides.pdf

```
:c.add crd,crs2 is RVC & crd & crs2 & cop0001=0x2 & cop1315=0x4 & cop1212=0x1 & cop0711!=0 & cop0206!=0
{
        crd = crd + crs2;
}
```

- You define there the register definitions, aliases, instructions etc.
- Ghidra can be compiled with a bad SLASPEC if its syntax is correct:
  - Then you will see on runtime if it works, or you will see tons of JAVA exceptions
  - We used "check & try" + "calm down" technique to achieve what we wanted :)

BRINGING RISC-V TO GHIDRA

```
define token instr (32)
  op0001=(0,1)
  op0204=(2,4)
  op0506=(5,6)
  op0707=(7,7)
  op0711=(7,11)
  r0711=(7,11)
  fr0711=(7,11)
  op0811=(8,11)
  op1214=(12,14)
  funct3=(12,14)
  op1219=(12,19)
  op1231=(12,31)
  sop1231=(12,31) signed
  op1519=(15,19)
  r1519=(15,19)
  fr1519=(15,19)
  op1527=(15,27)
  op1531=(15,31)
  op2020=(20,20)
  succ=(20,23)
  op2024=(20,24)
  r2024=(20,24)
  fr2024=(20,24)
  csr_0=(20,27)
```

rt?

sfully inte

```
define register offset=0x90000000 size=$(XLEN) [ ustatus ];
define register offset=0x90000010 size=$(XLEN) [ fflags ];
define register offset=0x90000020 size=$(XLEN) [ frm ];
define register offset=0x90000030 size=$(XLEN) [ fcsr ];
```

ng SLEIGH language to describe the CPU

a pro

```
attach variables [ csr_0 ]
          [ ustatus fflags frm fcsr uie utvec _ _ _ _ _ _ _ _ _ _
        _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
        _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
        _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
        uscratch uepc ucause utval uip _ _ _ _ _ _ _ _ _ _ _
        _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

he

docu

to m

ainful

was f

ready supported CPUs as a "source of knowledge"

ly, we found only one really useful resource - Guillaume Valadon presentation:

edou.github.io/talks/2019_BeeRump/slides.pdf

```
:c.add crd,crs2 is RVC & crd & crs2 & cop0001=0x2 & cop1315=0x4 & cop1212=0x1 & cop0711!=0 & cop0206!=0
{
        crd = crd + crs2;
}
```

```
# c.beqz Cs,Cp 0000c001 0000e003 CONDBRANCH (0, 0)
:c.beqz cr0709s,cbimm is RVC & cbimm & cr0709s & cop0001=0x1 & cop1315=0x6
{
    if (cr0709s == 0) goto cbimm;
}
```

- We used "check & try" + "calm down" technique to achieve what we wanted :)

BRINGING RISC-V TO GHIDRA

```
define token instr (32)
```

```
define register offset=0x90000000 size=$(XLEN) [ ustatus ];
```

```
13 05 00 00    mv      a0,zero
93 05 00 00    mv      a1,zero
13 06 00 00    mv      a2,zero
93 06 00 00    mv      a3,zero
13 07 00 00    mv      a4,zero
93 07 00 00    mv      a5,zero
13 08 00 00    mv      a6,zero
93 08 00 00    mv      a7,zero
13 09 00 00    mv      s2,zero
93 09 00 00    mv      s3,zero
13 0a 00 00    mv      s4,zero
93 0a 00 00    mv      s5,zero
13 0b 00 00    mv      s6,zero
93 0b 00 00    mv      s7,zero
13 0c 00 00    mv      s8,zero
93 0c 00 00    mv      s9,zero
13 0d 00 00    mv      s10,zero
93 0d 00 00    mv      s11,zero
13 0e 00 00    mv      t3,zero
93 0e 00 00    mv      t4,zero
13 0f 00 00    mv      t5,zero
93 0f 00 00    mv      t6,zero
97 f1 10 00    auipc   gp,0x10f
93 81 c1 32    addi    gp,gp,0x32c
               addi    sp,gp,0x400
               csrrwi  ter[redacted]
               auipc   a0,0x10f
               addi    a0,a0,-0x4e4
               addi    [redacted]
               bgeu    [redacted]

          LAB_[redacted]                    XREF[1]:     00080120(j)
               sd      [redacted]
               sd
```

```
19  }
20  if ((_DAT_0[redacted] & 6) == 0) {
21    unaff_retaddr = 0;
22    tp = 0;
23    register0[redacted] =
24    register0[redacted] =
25    puVar2 = (undefined8 *)&__data_start;
26    do {
27      *puVar2 = 0;
28      puVar2[1] = 0;
29      puVar2[2] = 0;
30      puVar2[3] = 0;
31      puVar2[4] = 0;
32      puVar2[5] = 0;
33      puVar2[6] = 0;
34      puVar2[7] = 0;
35      puVar2 = puVar2 + 8;
36    } while [redacted]
37    _ada_rv[redacted]
38  }
39  *(undefined8 *)((longlong)register0[redacted] + -8) = unaff_retaddr;
40  _DAT_0128165c = 0[redacted]
41  _DAT_012815b0 = 0[redacted]
42  _DAT_012815b8 = 0[redacted]0;
43  _DAT_012815bc = 0[redacted]
44  _DAT_012815b4 = 0[redacted]
45  _DAT_012815c0 = 0[redacted]
46  _DAT_012815c4 = 0[redacted]0;
47  [redacted]
48
49  uVar3 = 0xffffffffffffffff;
50  lVar5 = -1;
51  do {
52    lVar4 = lVar5 + 1;
```

```
:c.beqz cr0709s,cbimm is RVC & cbimm & cr0709s & cop0001=0x1 & cop1315=0x6
{
    if (cr0709s == 0) goto cbimm;
}
```

- We used "check & try" + "calm down" technique to achieve what we wanted :)

BRINGING RISC-V TO GHIDRA

```
define token instr (32)
```

```
define register offset=0x90000000 size=$(XLEN) [ ustatus ];
```

```
13 05 00 00    mv      a0,zero
93 05 00 00    mv      a1,zero
13 06 00 00    mv      a2,zero
93 06 00 00    mv      a3,zero
13 07 00 00    mv      a4,zero
93 07 00 00    mv      a5,zero
13 08 00 00    mv      a6,zero
93 08 00 00    mv      a7,zero
13 09 00 00    mv      s2,zero
93 09 00 00    mv      s3,zero
13 0a 00 00    mv      s4,zero
93 0a 00 00    mv      s5,zero
13 0b 00 00    mv      s6,zero
93 0b 00 00    mv      s7,zero
13 0c 00 00    mv      s8,zero
93 0c 00 00    mv      s9,zero
13 0d 00 00    mv      s10,zero
93 0d 00 00    mv      s11,zero
13 0e 00 00    mv      t3,zero
93 0e 00 00    mv      t4,zero
13 0f 00 00    mv      t5,zero
93 0f 00 00    mv      t6,zero
97 f1 10 00    auipc   gp,0x10f
93 81 c1 32    addi    gp,gp,0x32c
               addi    sp,gp,0x400
               csrrwi  ter[redacted]
               auipc   a0,0x10f
               addi    a0,a0,-0x4e4
               addi    [redacted]
               bgeu    [redacted]

LAB_[redacted]                    XREF[1]:    00080120(j)
               sd      [redacted]
               sd      [redacted]
```

```
19  }
20  if ((_DAT_0[redacted] & 6) == 0) {
21    unaff_retaddr = 0;
22    tp = 0;
23    register0[redacted] = [redacted]
24    register0[redacted] = [redacted]
25    puVar2 = (undefined8 *)&__data_start;
26    do {
27      *puVar2 = 0;
28      puVar2[1] = 0;
29      puVar2[2] = 0;
30      puVar2[3] = 0;
31      puVar2[4] = 0;
32      puVar2[5] = 0;
33      puVar2[6] = 0;
34      puVar2[7] = 0;
35      puVar2 = puVar2 + 8;
36    } while [redacted]
37    _ada_rv[redacted]
38  }
39  *(undefined8 *)((longlong)register0[redacted] + -8) = unaff_retaddr;
40  _DAT_0128165c = 0[redacted]
41  _DAT_012815b0 = 0[redacted]
42  _DAT_012815b8 = 0[redacted]0;
43  _DAT_012815bc = 0[redacted]
44  _DAT_012815b4 = 0[redacted]
45  _DAT_012815c0 = 0[redacted]
46  _DAT_012815c4 = 0[redacted]0;
47  [redacted]
48
49  uVar3 = 0xffffffffffffffff;
50  lVar5 = -1;
51  do {
52    lVar4 = lVar5 + 1;
```

```
:c.beqz cr0709s,cbimm is RVC & cbimm & cr0709s & cop0001=0x1 & cop1315=0x6
{
    if (cr0709s == 0) goto cbimm;
}
```

- We used "check & try" + "calm down" technique to achieve what we wanted :)

PROBLEM WITH MTVEC

- What to look for?
  - SPARK limits what we could hunt for…
  - We focused on the design and how HW is modeled
- We saw the very first instructions configuring the HW…
  - … and later setting up the MTVEC value
- What is MTVEC?
  - Official RISC-V documentation defines MTVEC register as a read-only or read/write register that holds the BASE address of the M-mode trap vector
  - By default, RISC-V handles all traps at any privilege level in machine mode (though a machine-mode handler might redirect traps back to the appropriate level)
  - When trap arrives, RISC-V switches to the machine mode and sets the instruction pointer counter (pc) register to the value configured in MTVEC.

- Wh
  -
  -
- We
  -
- Wh
  -
  -
  -

The `mtvec` register is an MXLEN-bit **WARL** read/write register that holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE).

MXLEN-1 … 2	1 … 0
BASE[MXLEN-1:2] (**WARL**)	MODE (**WARL**)
MXLEN-2	2

Figure 3.9: Machine trap-vector base-address register (`mtvec`).

The `mtvec` register must always be implemented, but can contain a hardwired read-only value. If `mtvec` is writable, the set of values the register may hold can vary by implementation. The value in the BASE field must always be aligned on a 4-byte boundary, and the MODE setting may impose additional alignment constraints on the value in the BASE field.

We allow for considerable flexibility in implementation of the trap vector base address. On the one hand, we do not wish to burden low-end implementations with a large number of state bits, but on the other hand, we wish to allow flexibility for larger systems.

Value	Name	Description
0	Direct	All exceptions set `pc` to BASE.
1	Vectored	Asynchronous interrupts set `pc` to BASE+4×cause.
≥2	—	*Reserved*

Table 3.5: Encoding of `mtvec` MODE field.

The encoding of the MODE field is shown in Table 3.5. When MODE=Direct, all traps into machine mode cause the `pc` to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into machine mode cause the `pc` to be set to the address in the BASE field, whereas interrupts cause the `pc` to be set to the address in the BASE field plus four times the interrupt cause number. For example, a machine-mode timer interrupt (see Table 3.6 on page 39) causes the `pc` to be set to BASE+`0x1c`.

y or
ector
mode (though
e level)
e instruction

The mtvec register is an MXLEN-bit **WARL** read/write register that holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE).

MXLEN-1 … 2	1 … 0
BASE[MXLEN-1:2] (WA…	MODE (**WARL**)
MX…	2

Figure 3.9: M… (mtvec).

The mtvec register must … d read-only value. If mtvec is writable, the se… ation. The value in the BASE field must al… tting may impose additional alignment c…

We allow for co… ss. On the one hand, we do… f state bits, but on the other…

Value	
0	
1	Ve… ause.
≥2	

The encoding of the MO… When MODE=Direct, all traps into machine mode cause the pc t… address in the BASE field. When MODE=Vectored, all synchronous exceptions into machine mode cause the pc to be set to the address in the BASE field, whereas interrupts cause the pc to be set to the address in the BASE field plus four times the interrupt cause number. For example, a machine-mode timer interrupt (see Table 3.6 on page 39) causes the pc to be set to BASE+0x1c.

- Wh…
  - …
  - …
- We…
  - …
- Wh…
  - …
  - …
  - …
  - …

y or
ector
mode (though
e level)
e instruction

- What will happen if any interrupt arrives before MTVEC is initialized?

PROBLEM WITH MTVEC

- RISC-V MTVEC register specifications does not define the initial value at all (undefined)
- We observed when the CPU starts, MTVEC is undefined by the standard though most of the tested implementations set it to 0
- In many implementations 0 is not a valid address (or not mapped) and any reference to it generates an exception
- If there is any trap/exception generated before initialization of MTVEC register, RISC-V ends up in a very "stable" infinitive exception loop
  - when exception arises, RISC-V reads MTVEC register (NULL value at that time) and tries to jump to the NULL page. This generates an exception again, because it's a reserved and not accessible memory, and it jumps to MTVEC again, and so on. RISC-V is not halted, it's just spinning in the infinitive exception loop.
- Such state is an ideal situation for a fault injection (glitching) attack. RISC-V is running at the highest privilege mode and constantly dereferencing glitchable register.

PROBLEM WITH MTVEC

- RISC-V MTVEC register specifications does not define the initial value at all (undefined)
- We observed when ned by the standard though most of the tested implen
- In many implementa not mapped) and any reference to it generates an ex
- If there is any trap/exception generated before initialization of MTVEC register, RISC-V ends up in a very "stable" infinitive exception loop
  - when exception arises, RISC-V reads MTVEC register (NULL value at that time) and tries to jump to the NULL page. This generates an exception again, because it's a reserved and not accessible memory, and it jumps to MTVEC again, and so on. RISC-V is not halted, it's just spinning in the infinitive exception loop.
- Such state is an ideal situation for a fault injection (glitching) attack. RISC-V is running at the highest privilege mode and constantly dereferencing glitchable register.

First bug:
ISA does not define the initial value of MTVEC register

PROBLEM WITH MTVEC

- RISC-V MTVEC register specifications does not define the initial value at all (undefined)
- We observed when ned by the standard though most of the tested implen
- In many implementa not mapped) and any reference to it generates an ex
- If there is any trap/exception generated before initialization of MTVEC register, RISC-V ends up in a very "stable" infinitive exception loop
  - when exception arises, RISC-V reads MTVEC register (NULL value at that time) and tries to jump to the NULL page. This generates an exception again, because it's a reserved and not accessible memory, and it jumps to MTVEC again, and so on. RISC-V is not halted, it's just spinning in the infinitive exception loop.
- Such state is an ideal C-V is running at the highest privile register.

First bug:
ISA does not define the initial value of MTVEC register

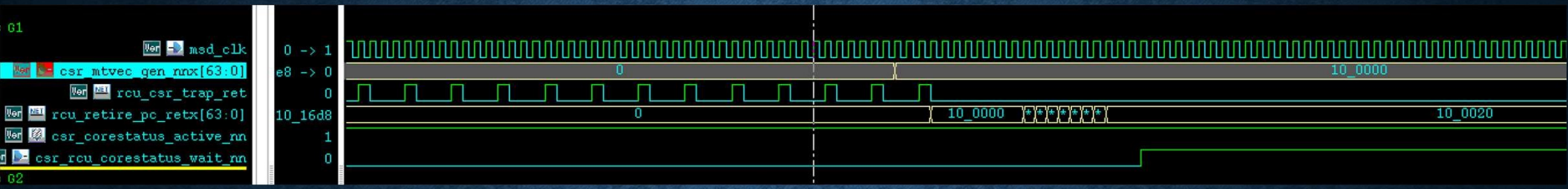
Second bug:
ISA „allows" for infinitive exception loop without halting the core (lack of „double/triple fault"-like exceptions)

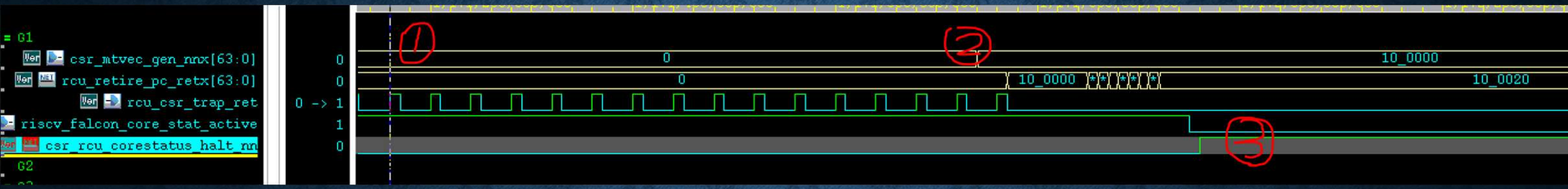
HOW TO EXPLOIT MTVEC?

- The described problem is fully exploitable if the attacker has the capabilities to:
  - Prefill D/I MEM of the RISC-V core (e.g., via „external" / recover (USB) boot functionality)
  - Generate an early exception during core execution (e.g., physical HW damage)
- Scenario:
  - Attacker pre-fills IMEM with the custom shellcode:
    - Attacker does that in a smart way by filling the entire IMEM with NOPs and in the edge of IMEM attacker puts a real shellcode.
  - Attacker boots RISC-V
  - Attacker enforces the necessary conditions to generate an early exception during Boot-SW or secure code execution and before MTVEC is initialized
  - RISC-V jumps to the NULL page and it enters the state of the infinitive exception loop (very stable and predictable state)
  - Attacker glitches the MTVEC register value of the looped core, and points it somewhere in the IMEM where special payload with the desired shellcode is placed (step 1):
    - Because MTVEC register has a NULL value, it is very likely that the change of just 1 bit ends up generating an address pointing in the middle of the NOPed filled IMEM memory.

HOW TO EXPLOIT MTVEC?

HOW TO EXPLOIT MTVEC?

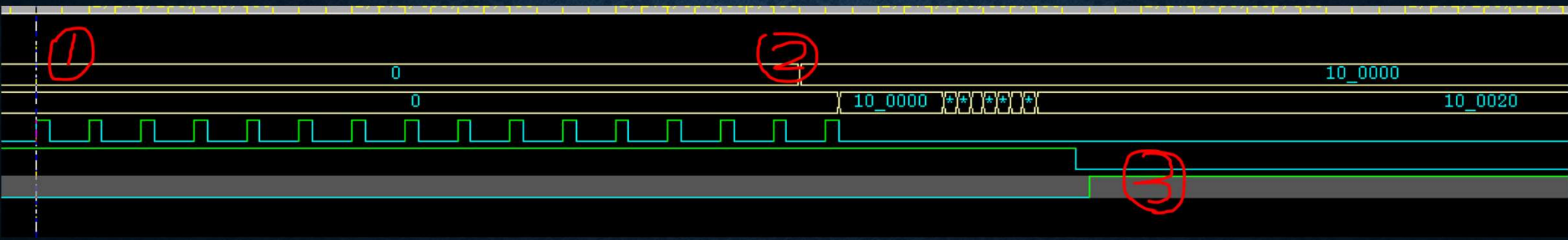




Step 1: pull a trigger to corrupt MTVEC register value on the looped core.

Step 2: the MTVEC value has been changed.

Step 3: ecall triggers the exception handler with the corrupted MTVEC.

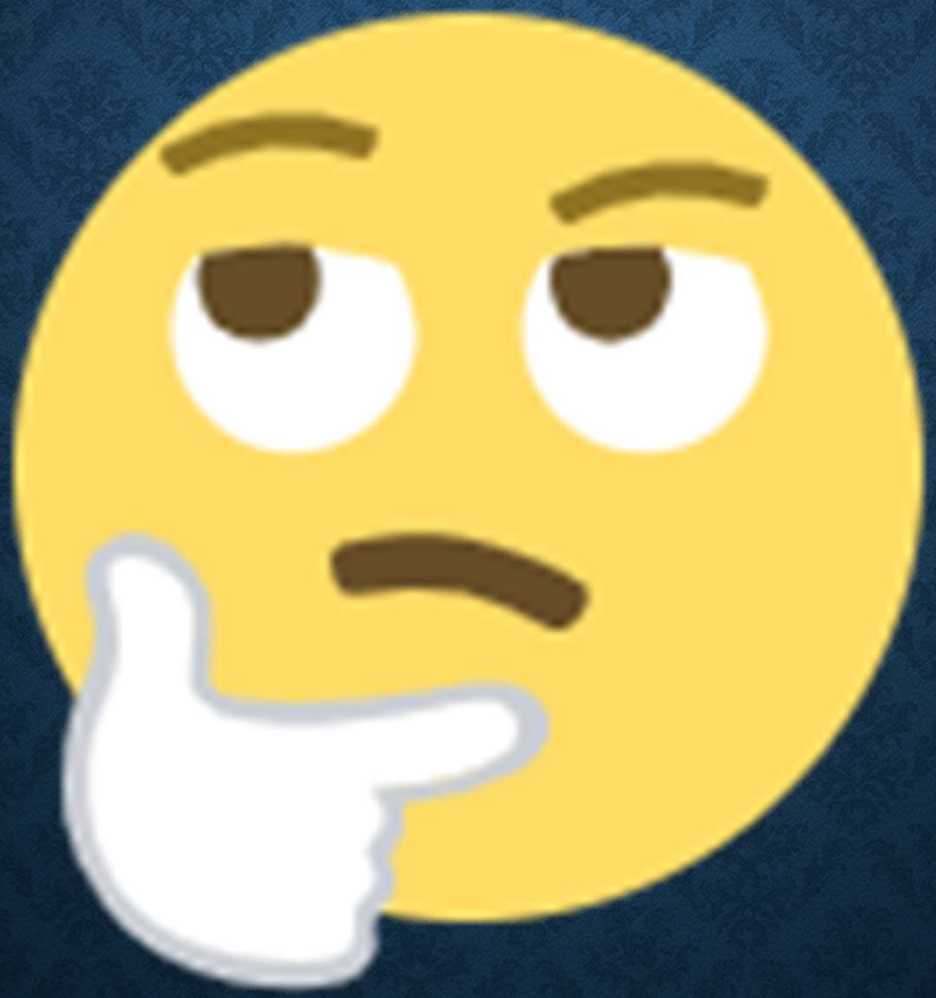


HOW TO EXPLOIT MTVEC?

- The described problem is fully exploitable if the attacker has the capabilities to:
  - Prefill D/I MEM of the RISC-V core (e.g., via „external" / recover (USB) boot functionality)
  - Generate an early exception during core execution (e.g., physical HW damage)
- Scenario:
  - Attacker pre-fill IMEM with the custom shellcode:
    - Attacker does it in a smart way by filling the entire IMEM with NOPs and in the edge of IMEM attacker puts a real shellcode.
  - Attacker boots RISC-V
  - Attacker [illegible] the [illegible] to generate [illegible] exception during Boot-SW or secure code execution and before MTVEC is initialized
  - RISC-V jumps to the NULL page and it enters the state of the infinitive exception loop (very stable and predictable state)
  - Attacker glitches the MTVEC register value of the looped core, and points it somewhere in the IMEM where special payload with the desired shellcode is placed (step 1):
    - Because MTVEC register has a NULL value, it is very likely that the change of just 1 bit ends up generating an address pointing in the middle of the NOPed filled IMEM memory.

DEMO

HOW TO REPORT AND FIX THE BUG IN ISA NOT IMPLEMENTATION?



HOW TO REPORT AND FIX THE BUG IN ISA NOT IMPLEMENTATION?

- The described problem(s) affects:
  - Uninitialized MTVEC:
    - All tested chips have MTVEC programmable (the most common mode) vulnerable to the described problem
    - Standard allows to have hardcoded read-only MTVEC value – in such case, it might point to the valid address (no bug)
  - Lack of "double/triple fault"-like exception
    - Standard doesn't define that at all – affects all the implementations

custom extension might fix that problems as well

- What did we do?
  - Contact RISC-V Foundation
    - Until that time, there was no official security response group – now there is one!
  - Contact SiFive
    - They were deeply involved in analyzing and working with the RISC-V Foundation to address the issue!
    - New CVE was allocated – **CVE-2021-1104**
  - Contact NVIDIA's internal RISC-V HW team
    - They confirmed and fixed the issue internally
    - Sync with all involved parties for responsible disclosure
- How to inform all the vendors (hundred+) about the issue(s)?
  - It can only be done through the RISC-V Foundation (with the SiFive help)

HOW TO FIX MTVEC ISSUE?

HOW TO FIX MTVEC ISSUE?

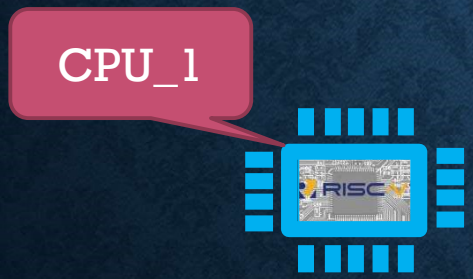
- The described problem is a chain of multiple problems…
  - To exploit the bug, we need to perform Fault Injection
- What are the effective Fault Injection protections?
  - DCLS (strong)
  - TCLS (even stronger!)
  - SW mitigation (complexity++)
  - Compiler mitigations

HOW TO FIX MTVEC ISSUE?

- The described problem is a chain of multiple problems…
  - To exploit the bug, we need to perform Fault Injection
- What are the effective Fault Injection protections?
  - DCLS (strong)
  - TCLS (even stronger!)
  - SW mitigation (complexity++)
  - Compiler mitigations

HOW TO FIX MTVEC ISSUE?

- The described problem is a chain of multiple problems…
  - To exploit the bug, we need to perform Fault Injection
- What are the effective Fault Injection protections?
  - DCLS (strong)
  - TCLS (even stronger!)
  - SW mitigation (complexity++)
  - Compiler mitigations

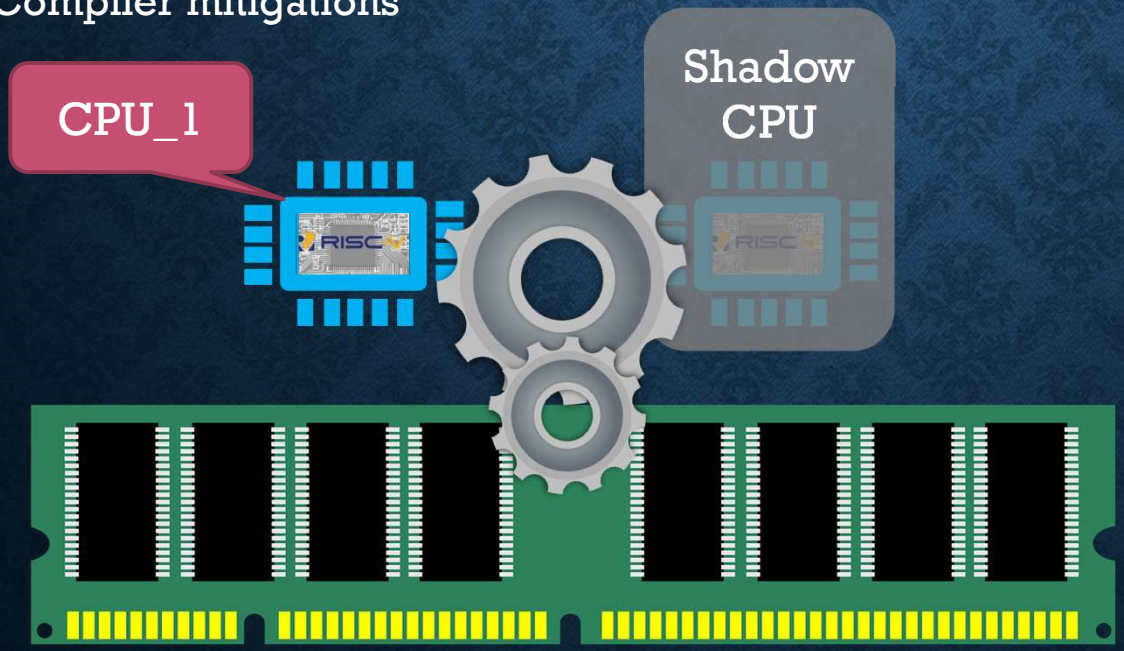


HOW TO FIX MTVEC ISSUE?

- The described problem is a chain of multiple problems…
  - To exploit the bug, we need to perform Fault Injection
- What are the effective Fault Injection protections?
  - DCLS (strong)
  - TCLS (even stronger!)
  - SW mitigation (complexity++)
  - Compiler mitigations

HOW TO FIX MTVEC ISSUE?

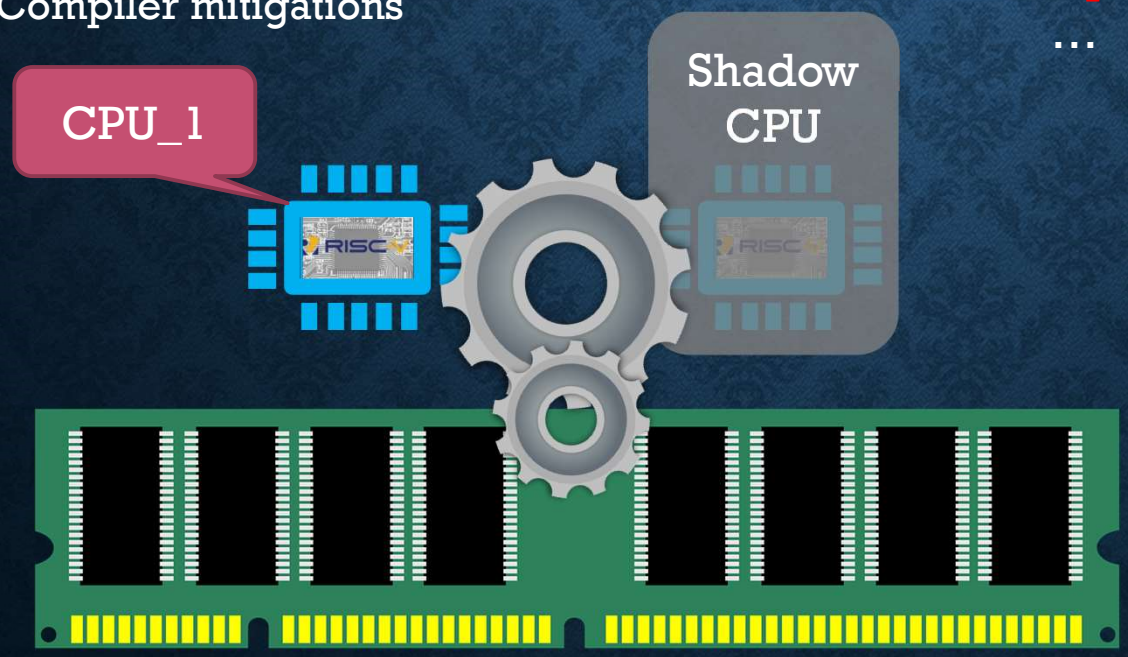
- The described problem is a chain of multiple problems…
  - To exploit the bug, we need to perform Fault Injection
- What are the effective Fault Injection protections?
  - DCLS (strong)
  - TCLS (even stronger!)
  - SW mitigation (complexity++)
  - Compiler mitigations



HOW TO FIX MTVEC ISSUE?

- The described problem is a chain of multiple problems…
  - To exploit the bug, we need to perform Fault Injection
- What are the effective Fault Injection protections?
  - DCLS (strong)
  - TCLS (even stronger!)
  - SW mitigation (complexity++)
  - Compiler mitigations

```
x = CPU_1(instruction_1)
y = Shadow_CPU(instruction_1)
if (x != y)
  panic();
…
```

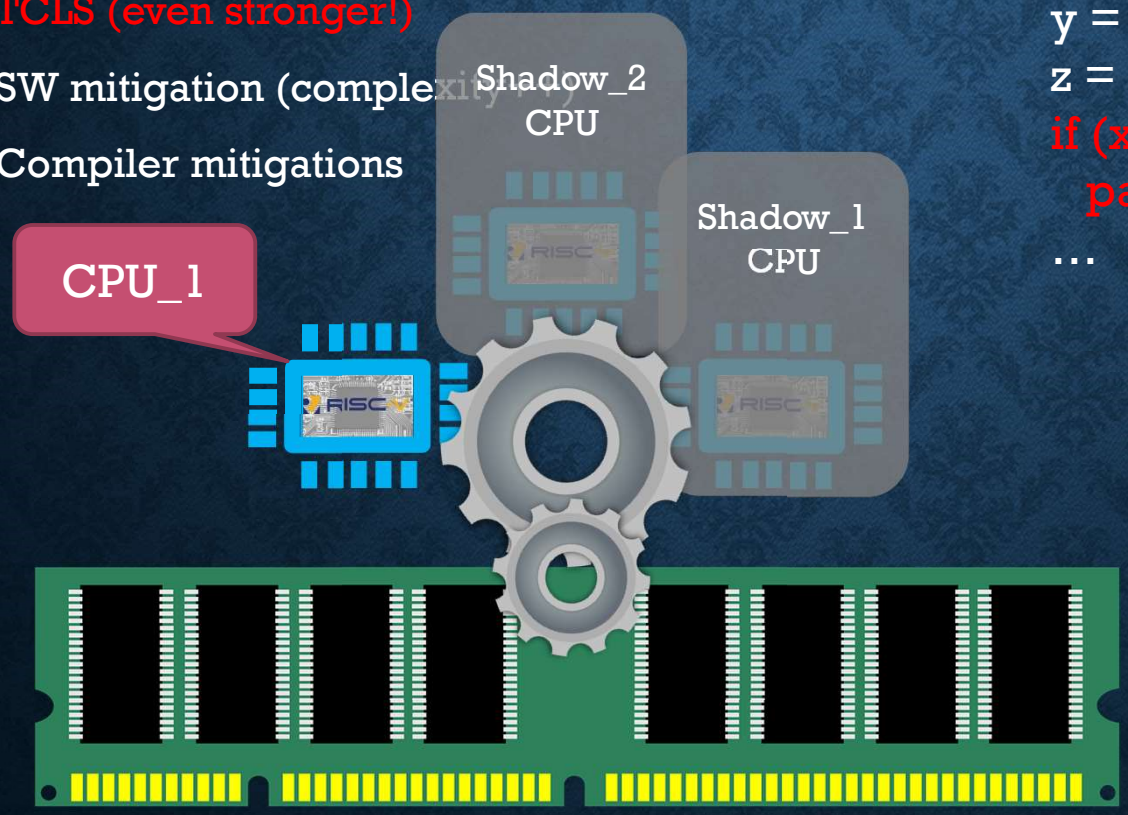


HOW TO FIX MTVEC ISSUE?

- The described problem is a chain of multiple problems…
  - To exploit the bug, we need to perform Fault Injection
- What are the effective Fault Injection protections?
  - DCLS (strong)
  - TCLS (even stronger!)
  - SW mitigation (complexity++)
  - Compiler mitigations

HOW TO FIX MTVEC ISSUE?

- The described problem is a chain of multiple problems…
  - To exploit the bug, we need to perform Fault Injection
- What are the effective Fault Injection protections?
  - DCLS (strong)
  - TCLS (even stronger!)
  - SW mitigation (complexity ↑↑)
  - Compiler mitigations




```
x = CPU_1(instruction_1)
y = Shadow_1_CPU(instruction_1)
z = Shadow_2_CPU(instruction_1)
if (x != y || x!=z || y!=z)
  panic();
…
```

HOW TO FIX MTVEC ISSUE?

- The described problem is a chain of multiple problems…
  - To exploit the bug, we need to perform Fault Injection
- What are the effective Fault Injection protections?
  - DCLS (strong)
  - TCLS (even stronger!)
  - SW mitigation (complexity++)
  - Compiler mitigations

HOW TO FIX MTVEC ISSUE?

- The described problem is a chain of multiple problems…
  - To exploit the bug, we need to perform Fault Injection
- What are the effective Fault Injection protections?
  - DCLS (strong)
  - TCLS (even stronger!)
  - SW mitigation (complexity++)
  - Compiler mitigations

1. Init/Re-init to fail/error
2. Branch re-check
3. Redundant checks
4. Pre-scrub payload destination
5. Clear memory on auth fail
6. Random delay
7. Exception on error (instead of inf. loop)
8. Hamming distance
9. Loop counter checks
10. Default fail
11. More…

HOW TO FIX MTVEC ISSUE?

- The described problem is a chain of multiple problems…
  - To exploit the bug, we need to perform Fault Injection
- What are the effective Fault Injection protections?
  - DCLS (strong)
  - TCLS (even stronger!)
  - SW mitigation (complexity++)
  - Compiler mitigations

Automatically applied by compiler

1. Init/Re-init to fail/error
2. Branch re-check
3. Redundant checks
4. Pre-scrub payload destination
5. Clear memory on auth fail
6. Random delay
7. Exception on error (instead of inf. loop)
8. Hamming distance
9. Loop counter checks
10. Default fail
11. More…

HOW TO FIX MTVEC ISSUE?

- The described problem is a chain of multiple problems…
  - To exploit the bug, we need to perform Fault Injection
- What are the effective Fault Injection protections?
  - DCLS (strong)
  - TCLS (even stronger!)
  - SW mitigation (complexity++)
  - Compiler mitigations

HOW TO FIX MTVEC ISSUE?

- The described problem is a chain of multiple problems…
  - To exploit the bug, we need to perform Fault Injection
- What are the effective Fault Injection protections?
  - DCLS (strong)
  - TCLS (even stronger!)
  - SW mitigation (complexity++)
  - Compiler mitigations
- Design decision to address MTVEC weakness:
  - As soon as START_CPU signal arrives, pre-initialize MTVEC to point to halt instruction
  - Change ISA to at least WARN about the potential problems with the late MTVEC initialization
  - Introduce "double / triple" fault-like exception which halts the core (instead of infinitive exception loop):
    - E.g., if MEPC == MTVEC then panic()

HOW TO FIX MTVEC ISSUE?

- The described problem is a chain of multiple problems…
  - To exploit the bug, we need to perform Fault Injection
- What are the effective Fault Injection protections?
  - DCLS (strong)
  - TCLS (even stronger!)
  - SW mitigation (complexity++)
  - Compiler mitigations
- Design decision to address MTVEC weakness:
  - As soon as START_CPU signal arrives, pre-initialize MTVEC to point to halt instruction
  - Change ISA to at least WARN about the potential problems with the late MTVEC initialization
  - Introduce "double / triple" fault-like exception which halts the core (instead of infinitive exception loop):
    - E.g., if MEPC == MTVEC then panic()
- What else can be done to harden RISC-V?
  - What about mitigation against the software attacks?

HARDENING RISC-V

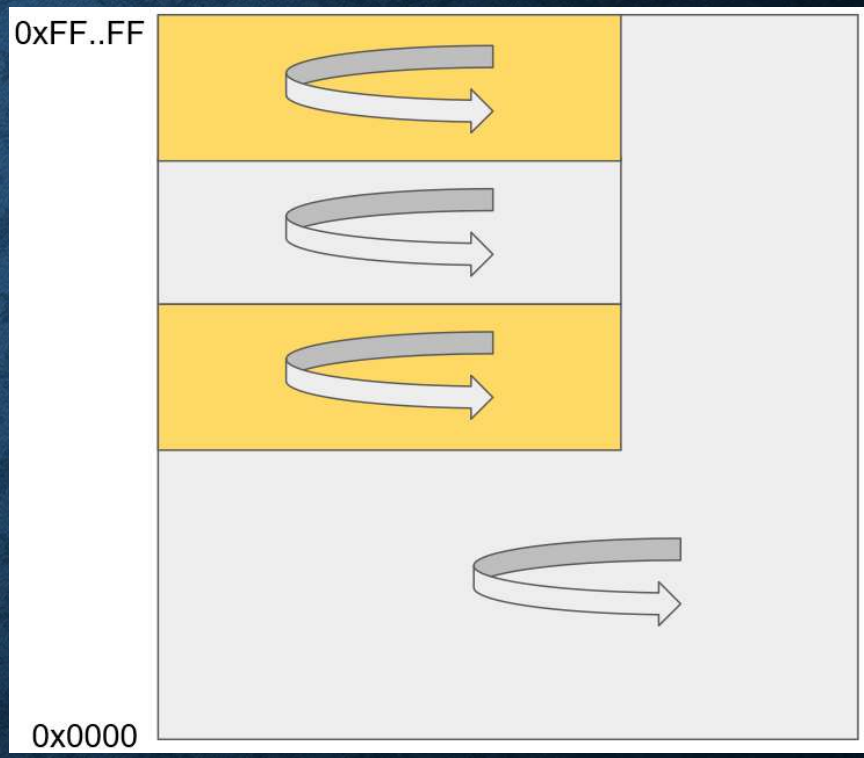
- Pointer Masking extension for RISC-V
  - Driven by Adam Zabrocki (NVIDIA), Martin Maas (Google), Lee Campbell (Google), RISC-V TEE and J-Ext Task Groups
  - From the security perspective it allows to implement:
    - HWASAN
    - Pointer Authentication Codes (PAC)
    - HW Memory Sandboxing
    - Foundation for:
      - HW MTE
      - Protecting RISC-V CFI (WIP)
      - Protecting RISC-V Shadow Stack (WIP)

HARDENING RISC-V

- Pointer Masking extension for RISC-V
  - Driven by Adam Zabrocki (NVIDIA), Martin Maas (Google), Lee Campbell (Google), RISC-V TEE and J-Ext Task Groups
  - From the security perspective it allows to implement:
    - HWASAN
    - Pointer Authentication Codes (PAC)
    - HW Memory Sandboxing
    - Foundation for:
      - HW MTE
      - Protecting RISC-V CFI (WIP)
      - Protecting RISC-V Shadow Stack (WIP)

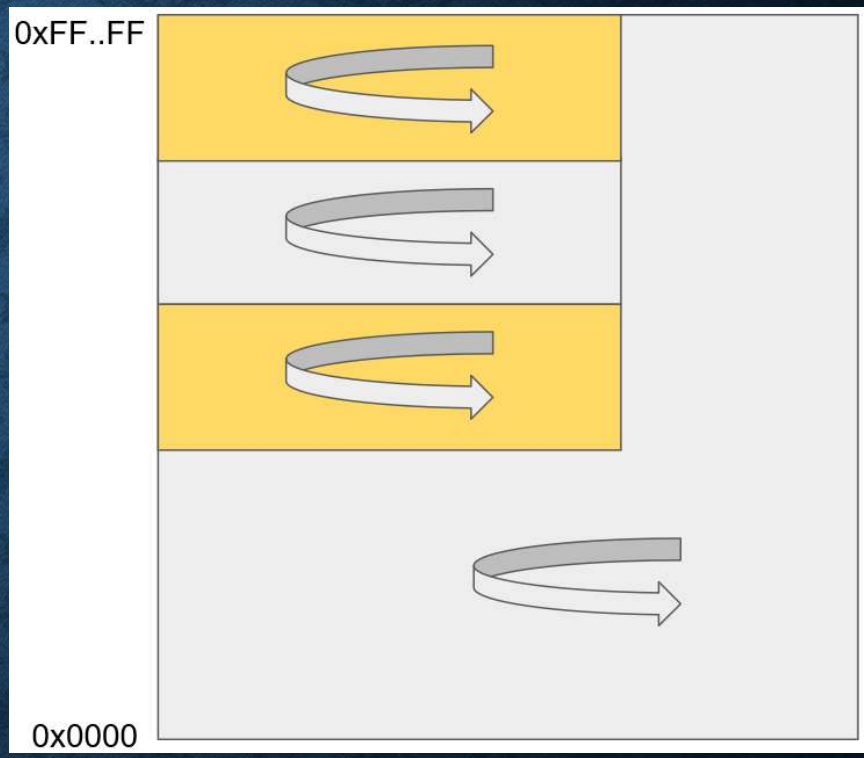
HARDENING RISC-V

- Pointer Masking extension for RISC-V
  - Driven by Adam Zabrocki (NVIDIA), Martin Maas (Google), Lee Campbell (Google), RISC-V TEE and J-Ext Task Groups
  - From the security perspective it allows to implement:
    - HWASAN
    - Pointer Authentication Codes (PAC)
    - HW Memory Sandboxing
    - Foundation for:
      - HW MTE
      - Protecting RISC-V CFI (WIP)
      - Protecting RISC-V Shadow Stack (WIP)

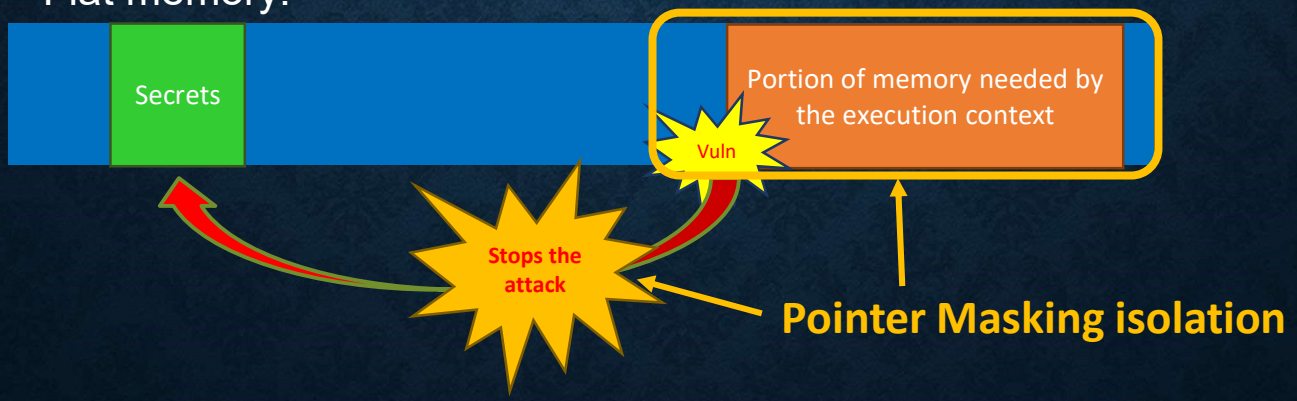


HARDENING RISC-V

- Pointer Masking extension for RISC-V
  - Driven by Adam Zabrocki (NVIDIA), Martin Maas (Google), Lee Campbell (Google), RISC-V TEE and J-Ext Task Groups
  - From the security perspective it allows to implement:
    - HWASAN
    - Pointer Authentication Codes (PAC)
    - HW Memory Sandboxing
    - Foundation for:
      - HW MTE
      - Protecting RISC-V CFI (WIP)
      - Protecting RISC-V Shadow Stack (WIP)

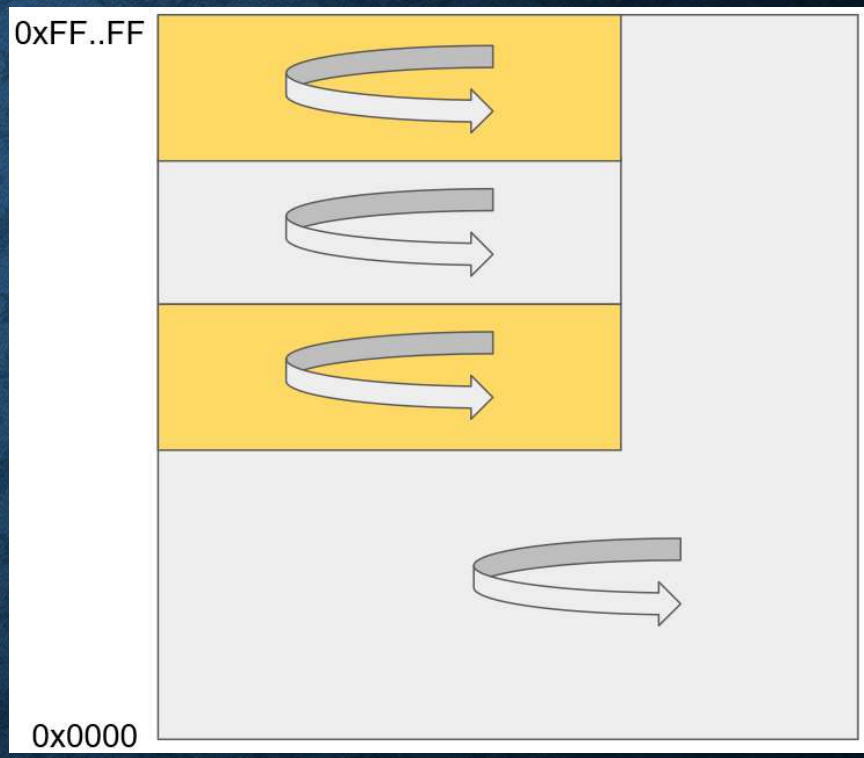


Flat memory:

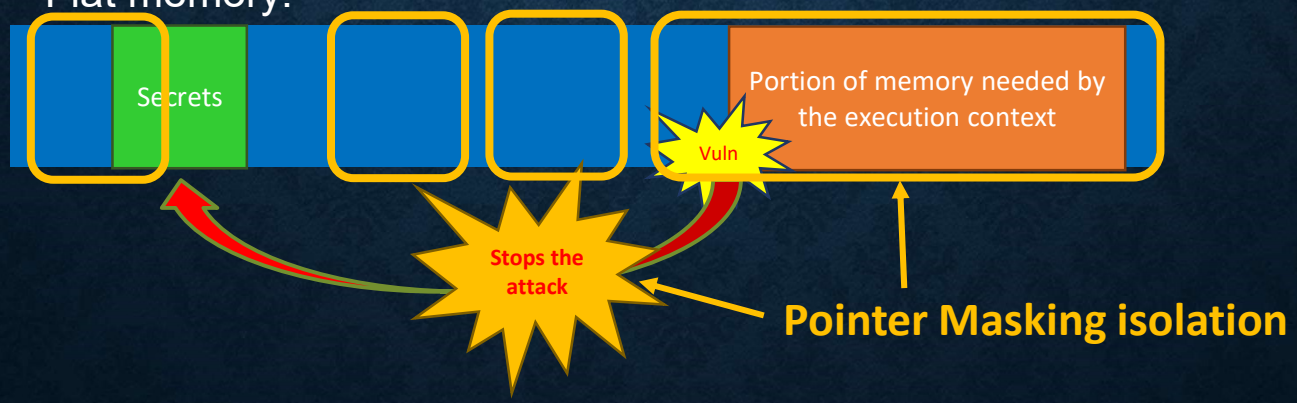


HARDENING RISC-V

- Pointer Masking extension for RISC-V
  - Driven by Adam Zabrocki (NVIDIA), Martin Maas (Google), Lee Campbell (Google), RISC-V TEE and J-Ext Task Groups
  - From the security perspective it allows to implement:
    - HWASAN
    - Pointer Authentication Codes (PAC)
    - HW Memory Sandboxing
    - Foundation for:
      - HW MTE
      - Protecting RISC-V CFI (WIP)
      - Protecting RISC-V Shadow Stack (WIP)



Flat memory:



ACKNOWLEDGMENTS

- We would like to thank:
  - NVIDIA:
    - GPU System Software:
      James Xu, Marko Mitic, Mateusz Kulikowski, RISC-V SW team
    - HW team:
      Joe Xie, Andy Ma, Jim Zhang, Dorin Yin, RISC-V HW team
    - Product Security:
      Alex Tereshkin, Shawn Richardson and PSIRT team
  - SiFive
  - RISC-V Foundation

SUMMARY

- The use of Type Safety languages and Formal Verification minimizes the attack surfaces for memory corruption issues, but it is not a silver bullet.

- There are CPU ISA bugs, and real-world attacks can combine physical attacks with software exploitation techniques.

- And the disclosure of ISA bugs is tough :-(

Q&A

Adam 'pi3' Zabrocki
Twitter: @Adam_pi3

Alex Matrosov
Twitter: @matrosov