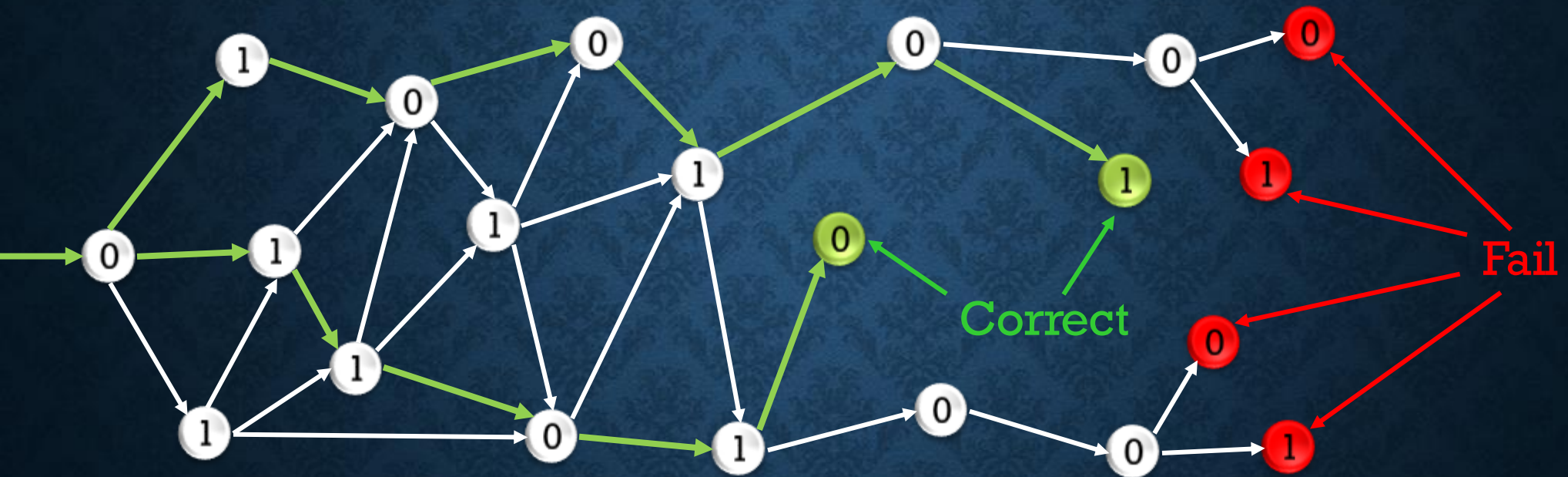


# EXPLOITATION IN THE ERA OF FORMAL VERIFICATION:



Adam 'pi3' Zabrocki  
Twitter: [@Adam\\_pi3](#)

Alex Tereshkin  
Twitter: [@AlexTereshkin](#)

## A PEEK AT A NEW FRONTIER WITH ADACORE/SPARK

# /USR/BIN/WHOWEARE



Private contact:

<http://pi3.com.pl>

[pi3@pi3.com.pl](mailto:pi3@pi3.com.pl)

Twitter: [@Adam\\_pi3](https://twitter.com/Adam_pi3)

Adam 'pi3' Zabrocki:

- Phrack author
- Bughunter (Hyper-V, KVM, RISC-V ISA, Intel, kernels, OpenSSH, Apache, more) – CVEs
- Creator and a developer of Linux Kernel Runtime Guard (LKRG)
- Speaker at BlackHat, DEF CON, BSides, Open-Source Tech and more
- The Pwnie Awards nominee



Private contact:

[alex.tereshkin@gmail.com](mailto:alex.tereshkin@gmail.com)

Twitter: [@AlexTereshkin](https://twitter.com/AlexTereshkin)

Alex Tereshkin:

- BlackHat, DEF CON speaker/trainer
- Reverse engineer
- UEFI security researcher
- ex-Invisible Things Lab researcher



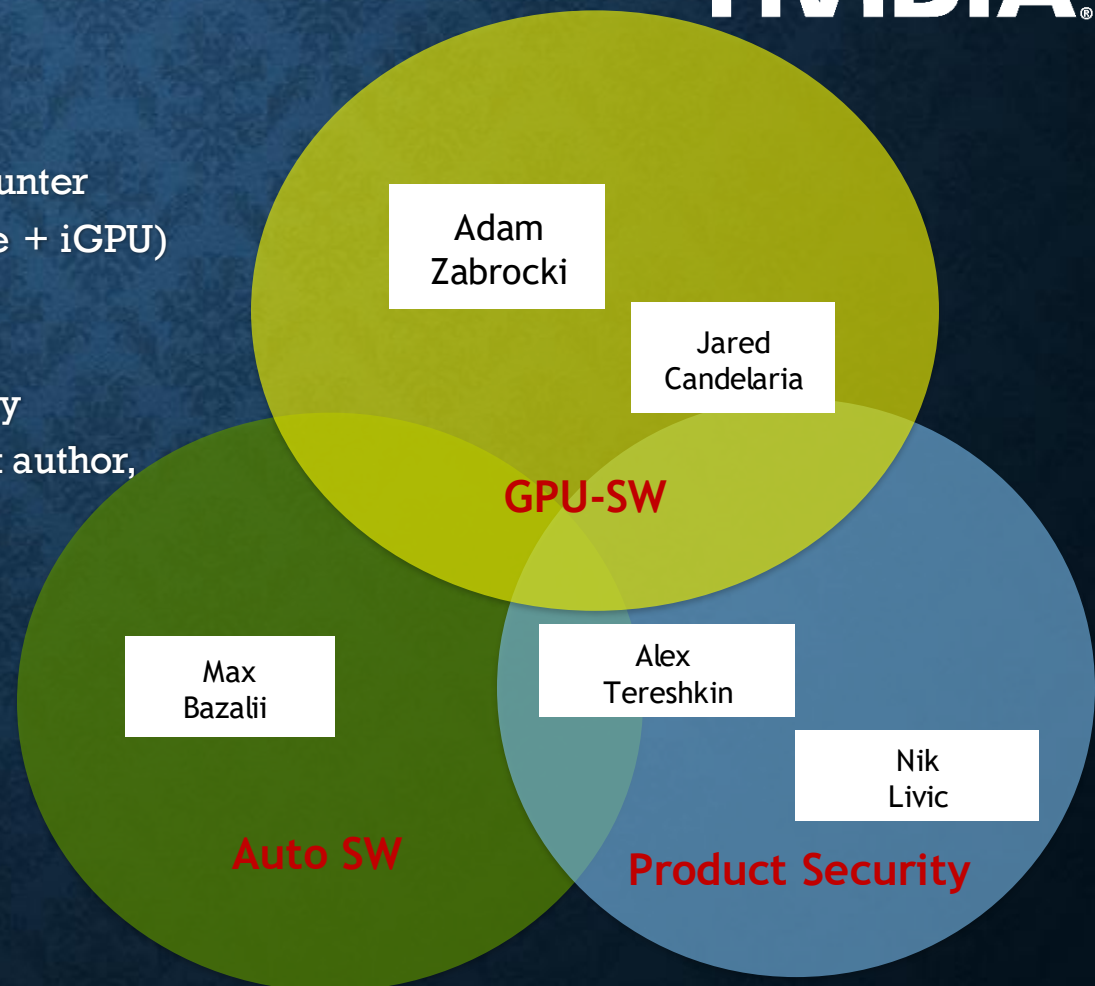


# OFFENSIVE SECURITY RESEARCH



NVIDIA®

- Adam 'pi3' Zabrocki (GPU SW) – leading
- Alex Tereshkin (Product Security)
- Jared Candelaria (GPU SW)  
Developer (Xbox One firmware, Hyper-V), Bughunter (WSL, Hyper-V, Stratix FPGA, Intel x86 Microcode + iGPU)
- Max Bazalii (Automotive)  
Reverse engineer since 2006, embedded security researcher since 2014. Apple iOS/tvOS jailbreak author, first public jailbreak for Apple WatchOS
- Nikola Livic (Product Security)  
Reversing, malware and vuln analysis and exploit dev since 2008



# MEMORY SAFETY PROBLEM



# MEMORY SAFETY PROBLEM

- ❖ What is "Memory Safety"?
- ❖ A term used by security engineers to describe a state of being protected from various bugs related to memory access

# MEMORY SAFETY PROBLEM

- ❖ What is "Memory Safety"?
  - ❖ A term used by security engineers to describe a state of being protected from various bugs related to memory access
- ❖ "Memory Safety" bugs include:
  - ❖ Buffer Overflow (Memory Overflow) including stack, heap, global, etc.
  - ❖ Out-of-bound read and/or write
  - ❖ Invalid Page Faults (including NULL pointer deref)
  - ❖ Use-After-Free, Use-After-Return, Use-After-Scope, etc.
  - ❖ Use of Uninit memory (including "wild pointers")
  - ❖ Memory leaks like stack/heap exhaustion, invalid free, etc.



# MEMORY SAFETY PROBLEM

- ❖ What is "Memory Safety"?
  - ❖ A term used by security engineers to describe a state of being protected from various bugs related to memory access
- ❖ "Memory Safety" bugs include:
  - ❖ Buffer Overflow (Memory Overflow) including stack, heap, global, etc.
  - ❖ Out-of-bound read and/or write
  - ❖ Invalid Page Faults (including NULL pointer deref)
  - ❖ Use-After-Free, Use-After-Return, Use-After-Scope, etc.
  - ❖ Use of Uninit memory (including "wild pointers")
  - ❖ Memory leaks like stack/heap exhaustion, invalid free, etc.
- ❖ "Memory Safety" does NOT include:
  - ❖ Integer overflow/underflow
  - ❖ Arithmetic overflow/underflow
  - ❖ Logical bugs
  - ❖ Error handling
  - ❖ Race conditions\*
  - ❖ Etc.

# MEMORY SAFETY PROBLEM

## ❖ What is "Memory Safety"?


- ❖ A term used by security engineers to describe a state of being protected from various bugs related to memory access

## ❖ "Memory Safety" bugs include:

- ❖ Buffer Overflow (Memory Overflow) including stack, heap, global, etc.
- ❖ Out-of-bound read and/or write
- ❖ Invalid Page Faults (including NULL pointer deref)
- ❖ Use-After-Free, Use-After-Return, Use-After-Scope, etc.
- ❖ Use of Uninit memory (including "wild pointers")
- ❖ Memory leaks like stack/heap exhaustion, invalid free, etc.

## ❖ "Memory Safety" does NOT include:

- ❖ Integer overflow/underflow
- ❖ Arithmetic overflow/underflow
- ❖ Logical bugs
- ❖ Error handling
- ❖ Race conditions\*
- ❖ Etc.



However, they often result in  
"Memory Safety" bugs  
(but not always)



# MEMORY SAFETY PROBLEM

- ❖ Why "Memory Safety"?
- ❖ Memory safety errors are today's biggest attack surface for attackers

# MEMORY SAFETY PROBLEM

- ❖ Why "Memory Safety"?
  - ❖ Memory safety errors are today's biggest attack surface for attackers
- ❖ "memory-unsafe" programming languages (like C/C++) are used to develop a core of the execution environment (e.g., Windows or Linux ecosystem)
  - ❖ allow developers fine-grained control of the memory addresses where their code can be executed.



# MEMORY SAFETY PROBLEM

- ❖ Why "Memory Safety"?
  - ❖ Memory safety errors are today's biggest attack surface for attackers
- ❖ "memory-unsafe" programming languages (like C/C++) are used to develop a core of the execution environment (e.g., Windows or Linux ecosystem)
  - ❖ allow developers fine-grained control of the memory addresses where their code can be executed.
- ❖ Memory safety bugs are very well researched...
  - ❖ "Professional exploiters" developed exploitation frameworks targeting specific software stack

# MEMORY SAFETY PROBLEM

- ❖ Why "Memory Safety"?
  - ❖ Memory safety errors are today's biggest attack surface for attackers
- ❖ "memory-unsafe" programming languages (like C/C++) are used to develop a core of the execution environment (e.g., Windows or Linux ecosystem)
  - ❖ allow developers fine-grained control of the memory addresses where their code can be executed.
- ❖ Memory safety bugs are very well researched...
  - ❖ "Professional exploiters" developed exploitation frameworks targeting specific software stack
- ❖ Many researchers focused on automation for memory safety bug detection without deep study of the target:
  - ❖ Example of that could be various code-coverage driven fuzzers



# MEMORY SAFETY PROBLEM

❖ Microsoft case:

# MEMORY SAFETY PROBLEM

## ❖ Microsoft case:

- ❖ In 2002, Bill Gates launched Microsoft's "Trustworthy Computing" initiative:

*"...had been under fire from some of its larger customers—government agencies, financial companies and others—about the security problems in Windows, issues that were being brought front and center by a series of self-replicating worms and embarrassing attacks."*



# MEMORY SAFETY PROBLEM

## ❖ Microsoft case:

- ❖ In 2002, Bill Gates launched Microsoft's "Trustworthy Computing" initiative:

*"...had been under fire from some of its larger customers—government agencies, financial companies and others—about the security problems in Windows, issues that were being brought front and center by a series of self-replicating worms and embarrassing attacks."*

## ❖ Focused on **Security**, Privacy, Reliability, and Business Integrity:

- ❖ Developed Security Development Lifecycle (SDLC)
- ❖ Created MSRC
- ❖ Adopted constant fuzzing, bughunting, exploit research, etc.

# MEMORY SAFETY PROBLEM

## ❖ Microsoft case:

- ❖ In 2002, Bill Gates launched Microsoft's "Trustworthy Computing" initiative:

*"...had been under fire from some of its larger customers—government agencies, financial companies and others—about the security problems in Windows, issues that were being brought front and center by a series of self-replicating worms and embarrassing attacks."*

- ❖ Focused on **Security**, Privacy, Reliability, and Business Integrity:

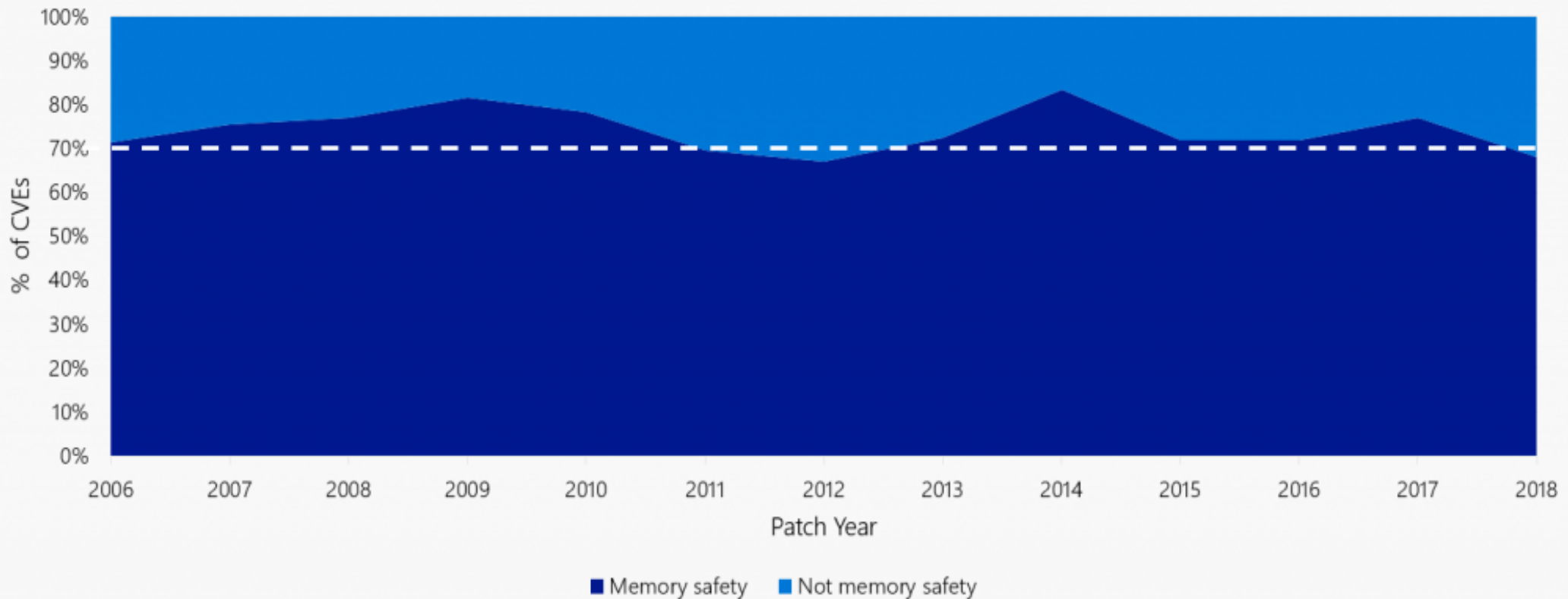
- ❖ Developed Security Development Lifecycle (SDLC)
- ❖ Created MSRC
- ❖ Adopted constant fuzzing, bughunting, exploit research, etc.

- ❖ In 2019, Microsoft analyzed the last 12 years of all reported security cases...



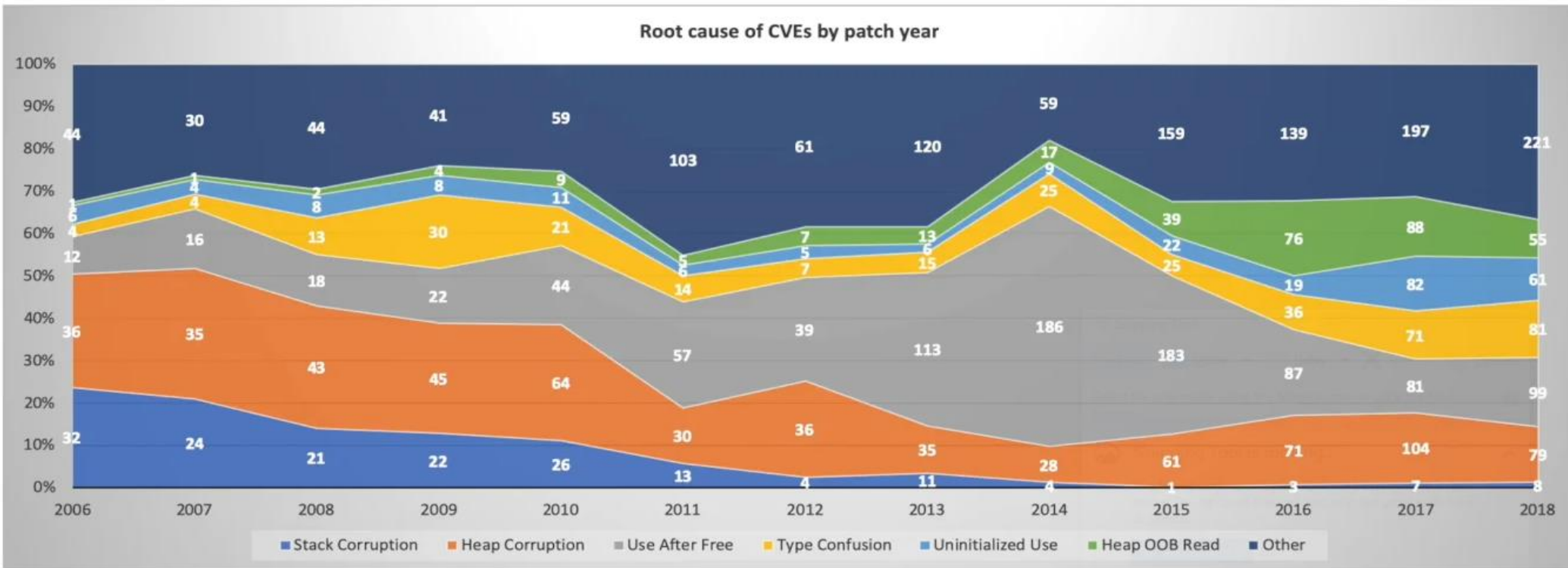
# MEMORY SAFETY PROBLEM

## ❖ Microsoft case:



# MEMORY SAFETY PROBLEM

## ❖ Microsoft case:



Top root causes since 2016:

#1: heap out-of-bounds

#2: use after free

#3: type confusion

#4: uninitialized use

[https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019\\_02\\_BlueHatIL/2019\\_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf](https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf)

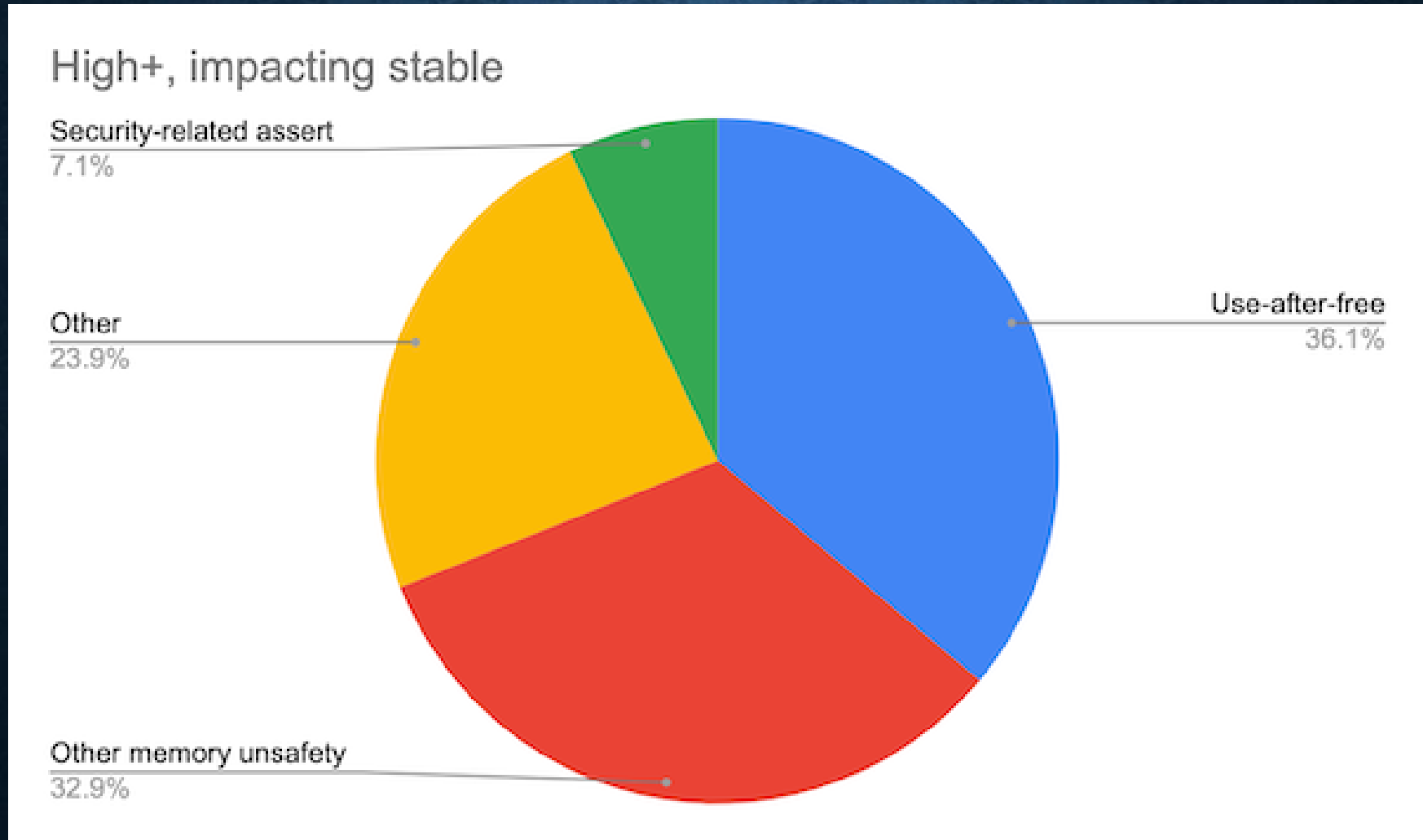


# MEMORY SAFETY PROBLEM

- ❖ Google Chrome case:
  - ❖ Design with Security in Mind
  - ❖ High code quality
  - ❖ Constant fuzzing since 2015(!):
    - ❖ By OSS-FUZZ platform:
      - ❖ Fuzzers: AFL, libFuzzer + tons of custom fuzzers
    - ❖ They use Google Cloud platform for fuzzing (essentially, unlimited computer power)
    - ❖ Google Chrome team itself have dedicated Chrome Security team
  - ❖ In 2020, Chrome team analyzed 912 security bugs (since 2015) with “high” and “critical” severity rating...

# MEMORY SAFETY PROBLEM

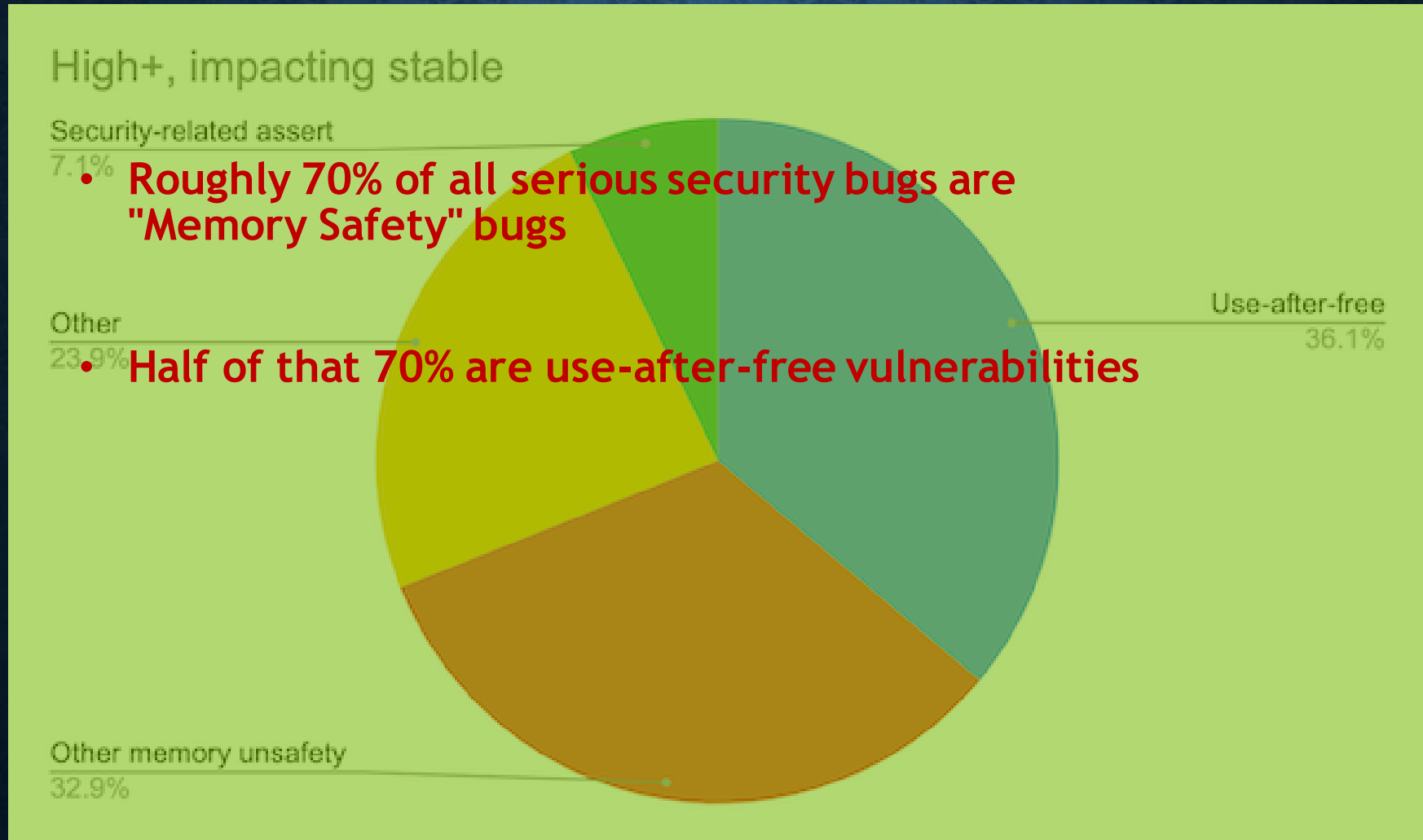
## ❖ Google Chrome case:





# MEMORY SAFETY PROBLEM

## ❖ Google Chrome case:



# MEMORY SAFETY PROBLEM

## ❖ Google Chrome case:

High+, impacting stable

Security-related assert

7.1%

- Roughly 70% of all serious security bugs are "Memory Safety" bugs

Other

23.9%

- Half of that 70% are use-after-free vulnerabilities

Use-after-free

36.1%

- Chromium's security architecture has always been designed to assume that these bugs exist, and code is sandboxed to stop them taking over the host machine

*"(...) But we are reaching the limits of sandboxing and site isolation."*

Other memory unsafety

32.9%



# MEMORY SAFETY PROBLEM

- ❖ Large study of vulnerability trends in OSS software over a decade performed by Technical University of Darmstadt (Germany), Continental AG and Intel Labs summarizes it as:

*“(...) we find no clear evidence that the vulnerability rate of widely used software decreases over time: Even in popular and “stable” releases, the fixing of bugs does not seem to reduce the rate of newly identified vulnerabilities.”*

<https://fileserver.tk.informatik.tu-armstadt.de/Publications/2020/alexopoulos2020TOPS.pdf>

# MEMORY SAFETY PROBLEM





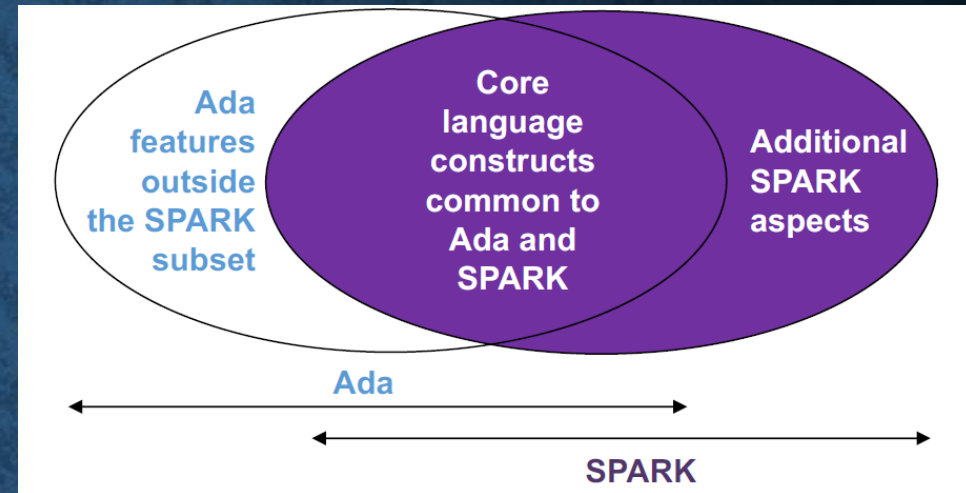
# MEMORY SAFETY PROBLEM



Formal verification techniques can prove the absence of memory safety and undefined behavior

# 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
- ❖ SPARK is safety certified





# WHAT IS ADACORE/SPARK?

- ❖ Programming language + set of analysis tools

- ❖ The strength is in the analysis tools...

- ❖ GNATProve, GNATStack, GNATTest, GNATEmulator

- You **can** compile buggy code – problems are detected by the tools and **developers might not run them at all**

- ❖ Proves that dynamic checks cannot fail

- ❖ Absence of Run-Time Errors

- ❖ Formal verification (Proofs)

- Tools are orthogonal and detect different classes of problems – to be fully protected **you must run all of them**

- ❖ Memory safe language (like RUST)

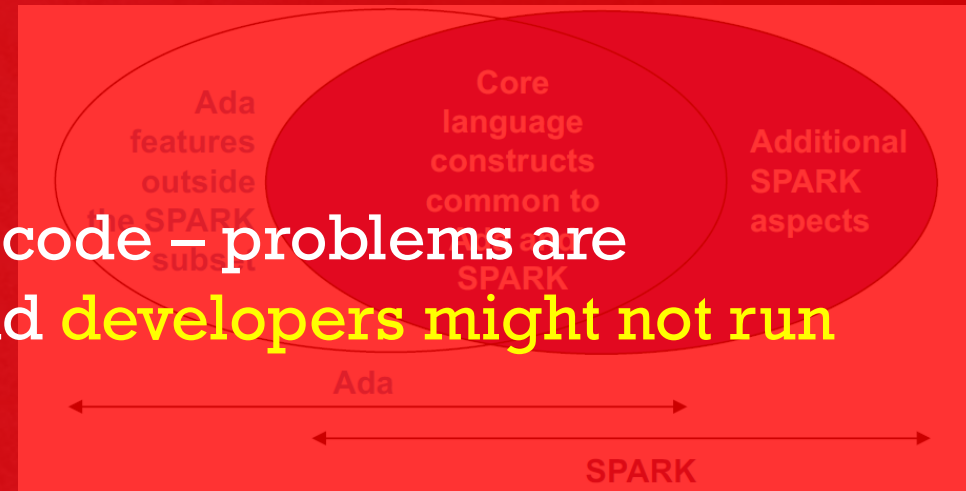
- ❖ Very strong typing system (much stronger than RUST)

- ❖ No arithmetic overflows, integer overflows, etc.

- ❖ Traditionally used in industries such as:

- ❖ Avionics, Railways, Defense, Auto, IoT

- ❖ SPARK is safety certified



# ADACORE/SPARK - EVALUATION

General memory corruption

Language	C	C++	SPARK
Type of Problem			
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

Language	C	C++	SPARK
Type of Problem			
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<sup>4</sup>

# ADACORE/SPARK - EVALUATION

General memory corruption

Language	C	C++	SPARK
Type of Problem			
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

Language	C	C++	SPARK
Type of Problem			
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 Pointer Arithmetic	Vulnerable	Might be safe (smart pointers)	N/A
Incorrect Pointer Scaling	Vulnerable	Might be safe (smart pointers)	N/A
Use of Pointer Subtraction to Dereference	Vulnerable	Might be limited in new standard but still possible	N/A
Assignment of a Fixed Address to a Pointer	Vulnerable	Vulnerable	N/A
Use of uninitialized memory	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

**SPARK relatively recently (~2019) introduced a concept of pointers using a concept similar to Rust's borrow checking:**

**<https://blog.adacore.com/pointer-based-data-structures-in-spark>**

\*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<sup>4</sup>



# ADACORE/SPARK - EVALUATION

## Arithmetic security

Language	C	C++	SPARK
Type of Problem			
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

Language	C	C++	SPARK
Type of Problem			
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

Language	C	C++	SPARK
Type of Problem			
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

Language	C	C++	SPARK
Type of Problem			
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

## ❖ Recap:

- ❖ 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 VS C/C++/...

- ❖ Everything sounds compelling but...

# ADACORE/SPARK VS C/C++/...

- ❖ Everything sounds compelling but...





# ADACORE/SPARK VS C/C++/...

- ❖ Everything sounds compelling but...
- ❖ Briefly about NVIDIA Offensive Security Research:
  - ❖ The main goal is to work as a "3rd party" independent and unbiased security team providing critical services
  - ❖ Proactively find security problems (known and unknown) before external researchers do
  - ❖ Identify new, novel and so far, unknown attack techniques
  - ❖ More...

# ADACORE/SPARK VS C/C++/...

- ❖ Everything sounds compelling but...
- ❖ Briefly about NVIDIA Offensive Security Research:
  - ❖ The main goal is to work as a "3rd party" independent and unbiased security team providing critical services
  - ❖ Proactively find security problems (known and unknown) before external researchers do
  - ❖ Identify new, novel and so far, unknown attack techniques
  - ❖ More...
- ❖ Some stats:
  - ❖ Since 2020 we've reviewed 17 high-impact projects
    - ❖ Memory unsafe language: 10
    - ❖ SPARK (4 in total):
      - ❖ Fully in SPARK: 2
      - ❖ SPARK as execution environment (and security enforcer): 2
    - ❖ Other: 3



# ADACORE/SPARK VS C/C++/...

- ❖ Everything sounds compelling but...
- ❖ Briefly about NVIDIA Offensive Security Research:
  - ❖ The main goal is to work as a "3rd party" independent and unbiased security team providing critical services
  - ❖ Proactively find security problems (known and unknown) before external researchers do
  - ❖ Identify new, novel and so far, unknown attack techniques
  - ❖ More...
- ❖ Some stats:
  - ❖ Since 2020 we've reviewed 17 high-impact projects
    - ❖ Memory unsafe language: 10
    - ❖ SPARK (4 in total):
      - ❖ Fully in SPARK: 2
      - ❖ SPARK as execution environment (and security enforcer): 2
    - ❖ Other: 3

# ADACORE/SPARK VS C/C++/...

❖ How to compare  to  not to  ?



# ADACORE/SPARK VS C/C++/...

❖ How to compare  to  not to  ?

❖ Raw data:

❖ Memory unsafe language:

	Timeframe	Total bugs	% of Mem Safety	Type
Project 1	5 weeks	37 (H:15 / M:8 / L:14)	66.7%	Virtualization component
Project 2	4 weeks	27 (H:3 / M:7 / L:17)	44%	RoT SW
Project 3	4 weeks	17 (H:9 / M:7 / L:1)	64.7%	Boot Control SW (NOT NVIDIA)
Project 4	3 weeks	45 (H:3 / M:23 / L:19)	52.2%	OS-like SW
Project 5	2 weeks	26 (H:13 / M:11 / L:2)	26.9%	System Control SW (NOT NVIDIA)
Project 6	< 2 weeks	13 (H:2 / M:9 / L:2)	57.1%	Resource Management FW / SW

❖ SPARK:

	Timeframe	Total bugs	SPARK only?	% of Mem Safety	Type
Project 1	Focused of HW modeling – Out Of Scope				
Project 2	6 weeks	10 (H:4 / M:4 / L:2)	No	0%	OS-like SW
Project 3 (Hybrid)	5 weeks	28 (H:9 / M:6 / L:13)	No	7.1%	RoT + Resource Management SW
Project 4	3-4 weeks	5 (H:2 / M:2 / L:1)	Yes	0%	Boot SW

# ADACORE/SPARK VS C/C++/...

❖ How to compare  to  not to  ?

❖ Raw data:

❖ Memory unsafe language:

	Timeframe	Total bugs	% of Mem Safety	Type
Project 1	5 weeks	37 (H:15 / M:8 / L:14)	66.7%	Virtualization component
Project 2	4 weeks	27 (H:3 / M:7 / L:17)	44%	RoT SW
Project 3	4 weeks	17 (H:9 / M:7 / L:1)	64.7%	Boot Control SW (NOT NVIDIA)
Project 4	3 weeks	45 (H:3 / M:23 / L:19)	52.2%	OS-like SW
Project 5	2 weeks	26 (H:13 / M:11 / L:2)	26.9%	System Control SW (NOT NVIDIA)
Project 6	< 2 weeks	13 (H:2 / M:9 / L:2)	57.1%	Resource Management FW / SW

❖ SPARK:

	Timeframe	Total bugs	SPARK only?	% of Mem Safety	Type
Project 1	Focused of HW modeling – Out Of Scope				
Project 2	6 weeks	10 (H:4 / M:4 / L:2)	No	0%	OS-like SW
Project 3 (Hybrid)	5 weeks	28 (H:9 / M:6 / L:13)	No	7.1%	RoT + Resource Management SW
Project 4	3-4 weeks	5 (H:2 / M:2 / L:1)	Yes	0%	Boot SW



# ADACORE/SPARK VS C/C++/...

❖ How to compare  to  not to  ?

❖ Raw data:

❖ Memory unsafe language:

	Timeframe	Total bugs	% of Mem Safety	Type
Project 1	5 weeks	37 (H:15 / M:8 / L:14)	66.7%	Virtualization component
Project 2	4 weeks	27 (H:3 / M:7 / L:17)	44%	RoT SW
Project 3	4 weeks	17 (H:9 / M:7 / L:1)	64.7%	Boot Control SW (NOT NVIDIA)
Project 4	3 weeks	45 (H:3 / M:23 / L:19)	52.2%	OS-like SW
Project 5	2 weeks	26 (H:13 / M:11 / L:2)	26.9%	System Control SW (NOT NVIDIA)
Project 6	< 2 weeks	13 (H:2 / M:9 / L:2)	57.1%	Resource Management FW / SW

❖ SPARK:

	Timeframe	Total bugs	SPARK only?	% of Mem Safety	Type
Project 1	Focused of HW modeling – Out Of Scope				
Project 2	6 weeks	10 (H:4 / M:4 / L:2)	No	0%	OS-like SW
Project 3 (Hybrid)	5 weeks	28 (H:9 / M:6 / L:13)	No	7.1%	RoT + Resource Management SW
Project 4	3-4 weeks	5 (H:2 / M:2 / L:1)	Yes	0%	Boot SW

# ADACORE/SPARK VS C/C++/...

❖ How to compare  to  not to  ?

❖ Raw data:

❖ Memory unsafe language:

	Timeframe	Total bugs	% of Mem Safety	Type
Project 1	5 weeks	37 (H:15 / M:8 / L:14)	66.7%	Virtualization component
Project 2	4 weeks	27 (H:3 / M:7 / L:17)	44%	RoT SW
Project 3	4 weeks	17 (H:9 / M:7 / L:1)	64.7%	Boot Control SW (NOT NVIDIA)
Project 4	3 weeks	45 (H:3 / M:23 / L:19)	52.2%	OS-like SW
Project 5	2 weeks	26 (H:13 / M:11 / L:2)	26.9%	System Control SW (NOT NVIDIA)
Project 6	< 2 weeks	13 (H:2 / M:9 / L:2)	57.1%	Resource Management FW / SW

❖ SPARK:

	Timeframe	Total bugs	SPARK only?	% of Mem Safety	Type
Project 1	Focused of HW modeling – Out Of Scope				
Project 2	6 weeks	10 (H:4 / M:4 / L:2)	No	0%	OS-like SW
Project 3 (Hybrid)	5 weeks	28 (H:9 / M:6 / L:13)	No	7.1%	RoT + Resource Management SW
Project 4	3-4 weeks	5 (H:2 / M:2 / L:1)	Yes	0%	Boot SW



# ADACORE/SPARK VS C/C++/...

❖ How to compare  to  not to  ?

❖ Raw data:

❖ Memory unsafe language:

	Timeframe	Total bugs	% of Mem Safety	Type
Project 1	5 weeks	37 (H:15 / M:8 / L:14)	66.7%	Virtualization component
Project 2	4 weeks	27 (H:3 / M:7 / L:17)	44%	RoT SW
Project 3	4 weeks	17 (H:9 / M:7 / L:1)	64.7%	Boot Control SW (NOT NVIDIA)
Project 4	3 weeks	45 (H:3 / M:23 / L:19)	52.2%	OS-like SW
Project 5	2 weeks	26 (H:13 / M:11 / L:2)	26.9%	System Control SW (NOT NVIDIA)
Project 6	< 2 weeks	13 (H:2 / M:9 / L:2)	57.1%	Resource Management FW / SW

❖ SPARK:

	Timeframe	Total bugs	SPARK only?	% of Mem Safety	Type
Project 1	Focused of HW modeling – Out Of Scope				
Project 2	6 weeks	10 (H:4 / M:4 / L:2)	No	0%	OS-like SW
Project 3 (Hybrid)	5 weeks	28 (H:9 / M:6 / L:13)	No	7.1%	RoT + Resource Management SW
Project 4	3-4 weeks	5 (H:2 / M:2 / L:1)	Yes	0%	Boot SW

# ADACORE/SPARK VS C/C++/...

❖ How to compare  to  not to  ?

❖ Raw data:

❖ Memory unsafe language:

	Timeframe	Total bugs	% of Mem Safety	Type
Project 1	5 weeks	37 (H:15 / M:8 / L:14)	66.7%	Virtualization component
Project 2	4 weeks	27 (H:3 / M:7 / L:17)	44%	RoT SW
Project 3	4 weeks	17 (H:9 / M:7 / L:1)	64.7%	Boot Control SW (NOT NVIDIA)
Project 4	3 weeks	45 (H:3 / M:23 / L:19)	52.2%	OS-like SW
Project 5	2 weeks	26 (H:13 / M:11 / L:2)	26.9%	System Control SW (NOT NVIDIA)
Project 6	< 2 weeks	13 (H:2 / M:9 / L:2)	57.1%	Resource Management FW / SW

❖ SPARK:

	Timeframe	Total bugs	SPARK only?	% of Mem Safety	Type
Project 1	Focused of HW modeling – Out Of Scope				
Project 2	6 weeks	10 (H:4 / M:4 / L:2)	No	0%	OS-like SW
Project 3 (Hybrid)	5 weeks	28 (H:9 / M:6 / L:13)	No	7.1%	RoT + Resource Management SW
Project 4	3-4 weeks	5 (H:2 / M:2 / L:1)	Yes	0%	Boot SW



# ADACORE/SPARK VS C/C++/...

❖ How to compare  to  not to  ?

❖ Raw data:

❖ Memory unsafe language:

	Timeframe	Total bugs	% of Mem Safety	Type
Project 1	5 weeks	37 (H:15 / M:8 / L:14)	66.7%	Virtualization component
Project 2	4 weeks	27 (H:3 / M:7 / L:17)	44%	RoT SW
Project 3	4 weeks	17 (H:9 / M:7 / L:1)	64.7%	Boot Control SW (NOT NVIDIA)
Project 4	3 weeks	45 (H:3 / M:23 / L:19)	52.2%	OS-like SW
Project 5	2 weeks	26 (H:13 / M:11 / L:2)	26.9%	System Control SW (NOT NVIDIA)
Project 6	< 2 weeks	13 (H:2 / M:9 / L:2)	57.1%	Resource Management FW / SW

❖ SPARK:

	Timeframe	Total bugs	SPARK only?	% of Mem Safety	Type
Project 1	Focused of HW modeling – Out Of Scope				
Project 2	6 weeks	10 (H:4 / M:4 / L:2)	No	0%	OS-like SW
Project 3 (Hybrid)	5 weeks	28 (H:9 / M:6 / L:13)	No	7.1%	RoT + Resource Management SW
Project 4	3-4 weeks	5 (H:2 / M:2 / L:1)	Yes	0%	Boot SW

# ADACORE/SPARK VS C/C++/...

❖ How to compare  to  not to  ?

❖ Raw data:

❖ Memory unsafe language:

	Timeframe	Total bugs	% of Mem Safety	Type
Project 1	5 weeks	37 (H:15 / M:8 / L:14)	66.7%	Virtualization component
Project 2	4 weeks	27 (H:3 / M:7 / L:17)	44%	RoT SW
Project 3	4 weeks	17 (H:9 / M:7 / L:1)	64.7%	Boot Control SW (NOT NVIDIA)
Project 4	3 weeks	45 (H:3 / M:23 / L:19)	52.2%	OS-like SW
Project 5	2 weeks	26 (H:13 / M:11 / L:2)	26.9%	System Control SW (NOT NVIDIA)
Project 6	< 2 weeks	13 (H:2 / M:9 / L:2)	57.1%	Resource Management FW / SW

❖ SPARK:

	Timeframe	Total bugs	SPARK only?	% of Mem Safety	Type
Project 1	Focused of HW modeling – Out Of Scope				
Project 2	6 weeks	10 (H:4 / M:4 / L:2)	No	0%	OS-like SW
Project 3 (Hybrid)	5 weeks	28 (H:9 / M:6 / L:13)	No	7.1%	RoT + Resource Management SW
Project 4	3-4 weeks	5 (H:2 / M:2 / L:1)	Yes	0%	Boot SW



# ADACORE/SPARK VS C/C++/...

❖ How to compare  to  not to  ?

❖ Raw data:

❖ Memory unsafe language:

	Timeframe	Total bugs	% of Mem Safety	Type
Project 1	5 weeks	37 (H:15 / M:8 / L:14)	66.7%	Virtualization component
Project 2	4 weeks	27 (H:3 / M:7 / L:17)	44%	RoT SW
Project 3	4 weeks	17 (H:9 / M:7 / L:1)	64.7%	Boot Control SW (NOT NVIDIA)
Project 4	3 weeks	45 (H:3 / M:23 / L:19)	52.2%	OS-like SW
Project 5	2 weeks	26 (H:13 / M:11 / L:2)	26.9%	System Control SW (NOT NVIDIA)
Project 6	< 2 weeks	13 (H:2 / M:9 / L:2)	57.1%	Resource Management FW / SW

❖ SPARK:

	Timeframe	Total bugs	SPARK only?	% of Mem Safety	Type
Project 1	Focused of HW modeling – Out Of Scope				
Project 2	6 weeks	10 (H:4 / M:4 / L:2)	No	0%	OS-like SW
Project 3 (Hybrid)	5 weeks	28 (H:9 / M:6 / L:13)	No	7.1%	RoT + Resource Management SW
Project 4	3-4 weeks	5 (H:2 / M:2 / L:1)	Yes	0%	Boot SW

# ADACORE/SPARK VS C/C++/...

❖ How to compare  to  not to  ?

❖ Raw data:

❖ Memory unsafe language:

	Timeframe	Total bugs	% of Mem Safety	Type
Project 1	5 weeks	37 (H:15 / M:8 / L:14)	66.7%	Virtualization component
Project 2	4 weeks	27 (H:3 / M:7 / L:17)	44%	RoT SW
Project 3	4 weeks	17 (H:9 / M:7 / L:1)	64.7%	Boot Control SW (NOT NVIDIA)
Project 4	3 weeks	45 (H:3 / M:23 / L:19)	52.2%	OS-like SW
Project 5	2 weeks	26 (H:13 / M:11 / L:2)	26.9%	System Control SW (NOT NVIDIA)
Project 6	< 2 weeks	13 (H:2 / M:9 / L:2)	57.1%	Resource Management FW / SW

❖ SPARK:

	Timeframe	Total bugs	SPARK only?	% of Mem Safety	Type
Project 1	Focused of HW modeling – Out Of Scope				
Project 2	6 weeks	10 (H:4 / M:4 / L:2)	No	0%	OS-like SW
Project 3 (Hybrid)	5 weeks	28 (H:9 / M:6 / L:13)	No	7.1%	RoT + Resource Management SW
Project 4	3-4 weeks	5 (H:2 / M:2 / L:1)	Yes	0%	Boot SW



# ADACORE/SPARK VS C/C++/...

❖ How to compare  to  not to  ?

❖ Raw data:

❖ Memory unsafe language:

	Timeframe	Total bugs	% of Mem Safety	Type
Project 1	5 weeks	37 (H:15 / M:8 / L:14)	66.7%	Virtualization component
Project 2	4 weeks	27 (H:3 / M:7 / L:17)	44%	RoT SW
Project 3	4 weeks	17 (H:9 / M:7 / L:1)	64.7%	Boot Control SW (NOT NVIDIA)
Project 4	3 weeks	45 (H:3 / M:23 / L:19)	52.2%	OS-like SW
Project 5	2 weeks	26 (H:13 / M:11 / L:2)	26.9%	System Control SW (NOT NVIDIA)
Project 6	< 2 weeks	13 (H:2 / M:9 / L:2)	57.1%	Resource Management FW / SW

❖ SPARK:

	Timeframe	Total bugs	SPARK only?	% of Mem Safety	Type
Project 1	Focused of HW modeling – Out Of Scope				
Project 2	6 weeks	10 (H:4 / M:4 / L:2)	No	0%	OS-like SW
Project 3 (Hybrid)	5 weeks	28 (H:9 / M:6 / L:13)	No	7.1%	RoT + Resource Management SW
Project 4	3-4 weeks	5 (H:2 / M:2 / L:1)	Yes	0%	Boot SW

# ADACORE/SPARK VS C/C++/...

❖ How to compare  to  not to  ?

❖ Raw data:

❖ Memory unsafe language:

	Timeframe	Total bugs	% of Mem Safety	Type
Project 1	5 weeks	37 (H:15 / M:8 / L:14)	66.7%	Virtualization component
Project 2	4 weeks	27 (H:3 / M:7 / L:17)	44%	RoT SW
Project 3	4 weeks	17 (H:9 / M:7 / L:1)	64.7%	Boot Control SW (NOT NVIDIA)
Project 4	3 weeks	45 (H:3 / M:23 / L:19)	52.2%	OS-like SW
Project 5	2 weeks	26 (H:13 / M:11 / L:2)	26.9%	System Control SW (NOT NVIDIA)
Project 6	< 2 weeks	13 (H:2 / M:9 / L:2)	57.1%	Resource Management FW / SW

❖ SPARK:

	Timeframe	Total bugs	SPARK only?	% of Mem Safety	Type
Project 1	Focused of HW modeling – Out Of Scope				
Project 2	6 weeks	10 (H:4 / M:4 / L:2)	No	0%	OS-like SW
Project 3 (Hybrid)	5 weeks	28 (H:9 / M:6 / L:13)	No	7.1%	RoT + Resource Management SW
Project 4	3-4 weeks	5 (H:2 / M:2 / L:1)	Yes	0%	Boot SW

# ADACORE/SPARK VS C/C++/...

- ❖ Conclusions (based on averaged data and SPARK language):
  - ❖ Formally verified software can be free from memory safety problems\*
  - ❖ Formally verified software has much higher quality because SPARK enforces:
    - ❖ Secure (and strict) coding practices
    - ❖ Strong typing
    - ❖ Correct init\*, data dependencies, coding violation, etc
    - ❖ You can't just sit and code (like in C/C++)... you are forced to “design” it upfront
  - ❖ SPARK may prove that dynamic checks cannot fail:
    - ❖ Absence of Run-Time Errors (AoRTE) – depends on levels of assurance (Silver+)
  - ❖ Enables more efficient offensive security efforts
    - ❖ Absence of “dummy” coding bugs
    - ❖ Unverified / unprovable access is clearly marked:
      - ❖ *“Unchecked\_Conversion” / “SPARK\_Mode => Off”*
    - ❖ Pre/Post-Conditions, Ghost code, etc – clearly defines the expected state
  - ❖ Most of the bugs which we saw requires deep knowledge and understanding of the software (more “deep” bugs, architecture problems, design bugs, etc):
    - ❖ OSR review of projects in memory unsafe languages – ~40-50 bugs in 4 weeks
    - ❖ OSR review of projects in SPARK – ~5-10 bugs in 6 weeks – better “quality” of bugs :)



# EXAMPLES OF REAL-WORLD BUGS



# EXAMPLES OF REAL-WORLD BUGS



# EXAMPLES OF REAL-WORLD BUGS

- ❖ Problem with Signature Verification



# EXAMPLES OF REAL-WORLD BUGS


## ❖ Problem with Signature Verification

```
xx1  procedure Verify_Public_Key(Err_Code: in out Error_Codes) is
...
xx3    Digest  : Sha_Digest;
xx4    Golden  : Sha_Digest;
xx5    Oem_Auk_Hoh: Nv_DW_Block;
xx6    Hash_Index : NvU32;
xx7    Oem_Fp    : NvU32;
xx8    Error    : Boolean;
...
x10  begin
x11    Do_some_verification(...
x12                                Err_Code=> Err_Code);
x13    Operation_Done:
x14    for Unused_Loop_Var in 1 .. 1 loop
x15
x16      if Verify_Config.Get_Auth = Config.AUTH_RSA then
x17
...
x70      } Verify_signature
x71    end if;
x72
x73  end loop Operation_Done;
x74
x75  Mark_operation_as_done(...
x76                                Err_Code=> Err_Code);
x77  end Verify_Public_Key;
```

# EXAMPLES OF REAL-WORLD BUGS

## ❖ Problem with Signature Verification


```
xx1  procedure Verify_Public_Key(Err_Code: in out Error_Codes) is
...
xx3    Digest : Sha_Digest;
xx4    Golden : Sha_Digest;
xx5    Oem_Auk_Hoh: Nv_DW_Block;
xx6    Hash_Index : NvU32;
xx7    Oem_Fp    : NvU32;
xx8    Error     : Boolean;
...
x10  begin
x11    Do_some_verification(...
x12                                Err_Code=> Err_Code);
x13    Operation_Done:
x14    for Unused_Loop_Var in 1 .. 1 loop
x15
x16      if Verify_Config.Get_Auth > Config.AUTH_RSA then
x17
...
x70
x71      end if;
x72
x73    end loop Operation_Done;
x74
x75    Mark_operation_as_done(...
x76                                Err_Code=> Err_Code);
x77  end Verify_Public_Key;
```



# EXAMPLES OF REAL-WORLD BUGS

## ❖ Problem with Signature Verification

```
xx1 procedure Verify_Public_Key(Err_Code: in out Error_Codes) is
...
xx3   Digest  : Sha_Digest;
xx4   Golden  : Sha_Digest;
xx5   Oem_Auk_Hoh: Nv_DW_Block;
xx6   Hash_Index : NvU32;
xx7   Oem_Fp    : NvU32;
xx8   Error     : Boolean;
...
x10 begin
x11   Do_some_verification(...
x12                           Err_Code=> Err_Code);
x13   Operation_Done:
x14   for Unused_Loop_Var in 1 .. 1 loop
x15
x16     if Verify_Config.Get_Auth > Config.AUTH_RSA then
x17
...
x70
x71   end if;
x72
x73   end loop Operation_Done;
x74
x75   Mark_operation_as_done(...
x76                           Err_Code=> Err_Code);
x77 end Verify_Public_Key;
```




```
1 function Get_Auth return Auth_Algorithm is
2 begin
3   return Result : Auth_Algorithm do
4     case Cpu.Get_REG_CTRL.Auth_Algo is
5       when NV_CPU_CTRL_REG_AUTH_ALGO_DISABLED =>
6         Result := AUTH_NONE;
7       when NV_CPU_CTRL_REG_AUTH_ALGO_RSA =>
8         Result := AUTH_RSA;
9       when others =>
10        Result := AUTH_UNKNOWN;
11     end case;
12   end return;
13
14 end Get_Auth;
```



# EXAMPLES OF REAL-WORLD BUGS

## ❖ Problem with Signature Verification

```
xx1 procedure Verify_Public_Key(Err_Code: in out Error_Codes) is
...
xx3   Digest  : Sha_Digest;
xx4   Golden  : Sha_Digest;
xx5   Oem_Auk_Hoh: Nv_DW_Block;
xx6   Hash_Index : NvU32;
xx7   Oem_Fp    : NvU32;
xx8   Error    : Boolean;
...
x10 begin
x11   Do_some_verification(...
x12                           Err_Code=> Err_Code);
x13   Operation_Done:
x14   for Unused_Loop_Var in 1 .. 1 loop
x15
x16     if Verify_Config.Get_Auth > Config.AUTH_RSA then
x17
...
x70
x71   end if;
x72
x73   end loop Operation_Done;
x74
x75   Mark_operation_as_done(...
x76                           Err_Code=> Err_Code);
x77 end Verify_Public_Key;
```



```
1 function Get_Auth return Auth_Algorithm is
2 begin
3   return Result : Auth_Algorithm do
4     case Cpu.Get_REG_CTRL.Auth_Algo is
5       when NV_CPU_CTRL_REG_AUTH_ALGO_DISABLED =>
6         Result := AUTH_NONE;
7       when NV_CPU_CTRL_REG_AUTH_ALGO_RSA =>
8         Result := AUTH_RSA;
9       when others =>
10        Result := AUTH_UNKNOWN;
11     end case,
12   end return;
13
14 end Get_Auth;
```

# EXAMPLES OF REAL-WORLD BUGS

## ❖ Problem with Signature Verification

```
xx1 procedure Verify_Public_Key(Err_Code: in out Error_Codes) is
...
xx3   Digest : Sha_Digest;
xx4   Golden : Sha_Digest;
xx5   Oem_Auk_Hoh: Nv_DW_Block;
xx6   Hash_Index : NvU32;
xx7   Oem_Fp : NvU32;
xx8   Error : Boolean;
...
x10 begin
x11   Do_some_verification(...
x12                           Err_Code=> Err_Code);
x13   Operation_Done:
x14   for Unused_Loop_Var in 1 .. 1 loop
x15
x16     if Verify_Config.Get_Auth > Config.AUTH_RSA then
x17
...
x70
x71   end if;
x72
x73   end loop Operation_Done;
x74
x75   Mark_operation_as_done(...
x76                           Err_Code=> Err_Code);
x77 end Verify_Public_Key;
```

**Verify\_signature**

```
1 function Get_Auth return Auth_Algorithm is
2 begin
3   return Result : Auth_Algorithm do
4     case Cpu.Get_REG_CTRL.Auth_Algo is
5       when NV_CPU_CTRL_REG_AUTH_ALGO_DISABLED =>
6         Result := AUTH_NONE;
7       when NV_CPU_CTRL_REG_AUTH_ALGO_RSA =>
8         Result := AUTH_RSA;
9       when others =>
10        Result := AUTH_UNKNOWN;
11     end case,
12   end return;
13
14 end Get_Auth;
```

#define NV_CPU_CTRL_REG_AUTH_ALGO	18:16
#define NV_CPU_CTRL_REG_AUTH_ALGO_DISABLED	0x00000000
#define NV_CPU_CTRL_REG_AUTH_ALGO_RSA	0x00000001

# EXAMPLES OF REAL-WORLD BUGS

## ❖ Problem with Signature Verification

```
xx1 procedure Verify_Public_Key(Err_Code: in out Error_Codes) is
...
xx3   Digest : Sha_Digest;
xx4   Golden : Sha_Digest;
xx5   Oem_Auk_Hoh: Nv_DW_Block;
xx6   Hash_Index : NvU32;
xx7   Oem_Fp : NvU32;
xx8   Error : Boolean;
...
x10 begin
x11   Do_some_verification(...
x12                           Err_Code=> Err_Code);
x13   Operation_Done:
x14   for Unused_Loop_Var in 1 .. 1 loop
x15
x16     if Verify_Config.Get_Auth > Config.AUTH_RSA then
x17
...
x70
x71   end if;
x72
x73   end loop Operation_Done;
x74
x75   Mark_operation_as_done(...
x76                           Err_Code=> Err_Code);
x77 end Verify_Public_Key;
```

**Verify\_signature**

```
1 function Get_Auth return Auth_Algorithm is
2 begin
3   return Result : Auth_Algorithm do
4     case Cpu.Get_REG_CTRL.Auth_Algo is
5       when NV_CPU_CTRL_REG_AUTH_ALGO_DISABLED =>
6         Result := AUTH_NONE;
7       when NV_CPU_CTRL_REG_AUTH_ALGO_RSA =>
8         Result := AUTH_RSA;
9       when others =>
10        Result := AUTH_UNKNOWN;
11     end case;
12   end return;
13
14 end Get_Auth;
```

3 bits == 8 states

**18:16**

```
#define NV_CPU_CTRL_REG_AUTH_ALGO
#define NV_CPU_CTRL_REG_AUTH_ALGO_DISABLED 0x00000000
#define NV_CPU_CTRL_REG_AUTH_ALGO_RSA 0x00000001
```



# EXAMPLES OF REAL-WORLD BUGS

## ❖ Problem with Signature Verification

```
xx1 procedure Verify_Public_Key(Err_Code: in out Error_Codes) is
...
xx3   Digest  : Sha_Digest;
xx4   Golden  : Sha_Digest;
xx5   Oem_Auk_Hoh: Nv_DW_Block;
xx6   Hash_Index : NvU32;
xx7   Oem_Fp    : NvU32;
xx8   Error    : Boolean;
...
x10 begin
x11   Do_some_verification(...
x12                           Err_Code=> Err_Code);
x13   Operation_Done:
x14   for Unused_Loop_Var in 1 .. 1 loop
x15
x16     if Verify_Config.Get_Auth > Config.AUTH_RSA then
x17
...
x70
x71     end if;
x72
x73   end loop Operation_Done;
x74
x75   Mark_operation_as_done(...
x76                           Err_Code=> Err_Code);
x77 end Verify_Public_Key;
```

**Verify\_signature**

```
1 function Get_Auth return Auth_Algorithm is
2 begin
3   return Result : Auth_Algorithm do
4     case Cpu.Get_REG_CTRL.Auth_Algo is
5       when NV_CPU_CTRL_REG_AUTH_ALGO_DISABLED =>
6         Result := AUTH_NONE;
7       when NV_CPU_CTRL_REG_AUTH_ALGO_RSA =>
8         Result := AUTH_RSA;
9       when others =>
10        Result := AUTH_UNKNOWN;
11     end case;
12   end return;
13
14 end Get_Auth;
```

6 states gets  
**AUTH\_UNKNOWN**

3 bits == 8 states

```
#define NV_CPU_CTRL_REG_AUTH_ALGO           18:16
#define NV_CPU_CTRL_REG_AUTH_ALGO_DISABLED  0x00000000
#define NV_CPU_CTRL_REG_AUTH_ALGO_RSA       0x00000001
```

# EXAMPLES OF REAL-WORLD BUGS

## ❖ Problem with Signature Verification

```
xx1 procedure Verify_Public_Key(Err_Code: in out Error_Codes) is
...
xx3   Digest : Sha_Digest;
xx4   Golden : Sha_Digest;
xx5   Oem_Auk_Hoh: Nv_DW_Block;
xx6   Hash_Index : NvU32;
xx7   Oem_Fp : NvU32;
xx8   Error : Boolean;
...
x10 begin
x11   Do_some_verification(...
x12                           Err_Code=> Err_Code);
x13   Operation_Done:
x14   for Unused_Loop_Var in 1 .. 1 loop
x15
x16     if Verify_Config.Get_Auth > Config.AUTH_RSA then
x17
...
x70
x71   end if;
x72
x73   end loop Operation_Done;
x74
x75   Mark_operation_as_done(...
x76                           Err_Code=> Err_Code);
x77 end Verify_Public_Key;
```

**Verify\_signature**

```
1 function Get_Auth return Auth_Algorithm is
2 begin
3   return Result : Auth_Algorithm do
4     case Cpu.Get_REG_CTRL.Auth_Algo is
5       when NV_CPU_CTRL_REG_AUTH_ALGO_DISABLED =>
6         Result := AUTH_NONE;
7       when NV_CPU_CTRL_REG_AUTH_ALGO_RSA =>
8         Result := AUTH_RSA;
9       when others =>
10        Result := AUTH_UNKNOWN;
11     end case;
12   end return;
13
14 end Get_Auth;
```

6 states gets  
**AUTH\_UNKNOWN**

3 bits == 8 states

```
#define NV_CPU_CTRL_REG_AUTH_ALGO
#define NV_CPU_CTRL_REG_AUTH_ALGO_DISABLED 0x00000000
#define NV_CPU_CTRL_REG_AUTH_ALGO_RSA 0x00000001
```

**18:16**

# EXAMPLES OF REAL-WORLD BUGS

## ❖ Problem with Signature Verification

```
xx1 procedure Verify_Public_Key(Err_Code: in out Error_Codes) is
...
xx3   Digest  : Sha_Digest;
xx4   Golden  : Sha_Digest;
xx5   Oem_Auk_Hoh: Nv_DW_Block;
xx6   Hash_Index : NvU32;
xx7   Oem_Fp    : NvU32;
xx8   Error    : Boolean;
...
x10 begin
x11   Do_some_verification(...
x12                           Err_Code=> Err_Code);
x13   Operation_Done:
x14   for Unused_Loop_Var in 1 .. 1 loop
x15
x16     if Verify_Config.Get_Auth > Config.AUTH_RSA then
x17
...
x70
x71   end if;
x72
x73   end loop Operation_Done;
x74
x75   Mark_operation_as_done(...
x76                           Err_Code=> Err_Code);
x77 end Verify_Public_Key;
```

**Verify\_signature**

```
1 function Get_Auth return Auth_Algorithm is
2 begin
3   return Result : Auth_Algorithm do
4     case Cpu.Get_REG_CTRL.Auth_Algo is
5       when NV_CPU_CTRL_REG_AUTH_ALGO_DISABLED =>
6         Result := AUTH_NONE;
7       when NV_CPU_CTRL_REG_AUTH_ALGO_RSA =>
8         Result := AUTH_RSA;
9       when others =>
10        Result := AUTH_UNKNOWN;
11     end case;
12   end return;
13
14 end Get_Auth;
```

6 states gets  
**AUTH\_UNKNOWN**

3 bits == 8 states

```
#define NV_CPU_CTRL_REG_AUTH_ALGO           18:16
#define NV_CPU_CTRL_REG_AUTH_ALGO_DISABLED 0x00000000
#define NV_CPU_CTRL_REG_AUTH_ALGO_RSA      0x00000001
```



# EXAMPLES OF REAL-WORLD BUGS

## ❖ Problem with Signature Verification

```
xx1 procedure Verify_Public_Key(Err_Code: in out Error_Codes) is
...
xx3   Digest  : Sha_Digest;
xx4   Golden  : Sha_Digest;
xx5   Oem_Auk_Hoh: Nv_DW_Block;
xx6   Hash_Index : NvU32;
xx7   Oem_Fp    : NvU32;
xx8   Error    : Boolean;
...
x10 begin
x11   Do_some_verification(...
x12                           Err_Code=> Err_Code);
x13   Operation_Done:
x14   for Unused_Loop_Var in 1 .. 1 loop
x15
x16     if Verify_Config.Get_Auth > Config.AUTH_RSA then
x17
...
x70
x71   end if;
x72
x73   end loop Operation_Done;
x74
x75   Mark_operation_as_done(...
x76                           Err_Code=> Err_Code);
x77 end Verify_Public_Key;
```

**Verify\_signature**

```
1 function Get_Auth return Auth_Algorithm is
2 begin
3   return Result : Auth_Algorithm do
4     case Cpu.Get_REG_CTRL.Auth_Algo is
5       when NV_CPU_CTRL_REG_AUTH_ALGO_DISABLED =>
6         Result := AUTH_NONE;
7       when NV_CPU_CTRL_REG_AUTH_ALGO_RSA =>
8         Result := AUTH_RSA;
9       when others =>
10        Result := AUTH_UNKNOWN;
11     end case;
12   end return;
13
14 end Get_Auth;
```

6 states gets  
**AUTH\_UNKNOWN**

3 bits == 8 states

```
#define NV_CPU_CTRL_REG_AUTH_ALGO           18:16
#define NV_CPU_CTRL_REG_AUTH_ALGO_DISABLED  0x00000000
#define NV_CPU_CTRL_REG_AUTH_ALGO_RSA       0x00000001
```

Out of 8 states only 1 enforces signature verification and 7 states are handled as no verification:

- Prover didn't catch that
- It's a logical error

# EXAMPLES OF REAL-WORLD BUGS

- ❖ Problem with the compiler weakening FI protection

# EXAMPLES OF REAL-WORLD BUGS

- ❖ Problem with the compiler weakening FI protection

During code review, we observed redundant constant time loops to check equality for FI countermeasure robustness.

Additionally, we saw that the Hamming distance between pass and not pass states, was shortened

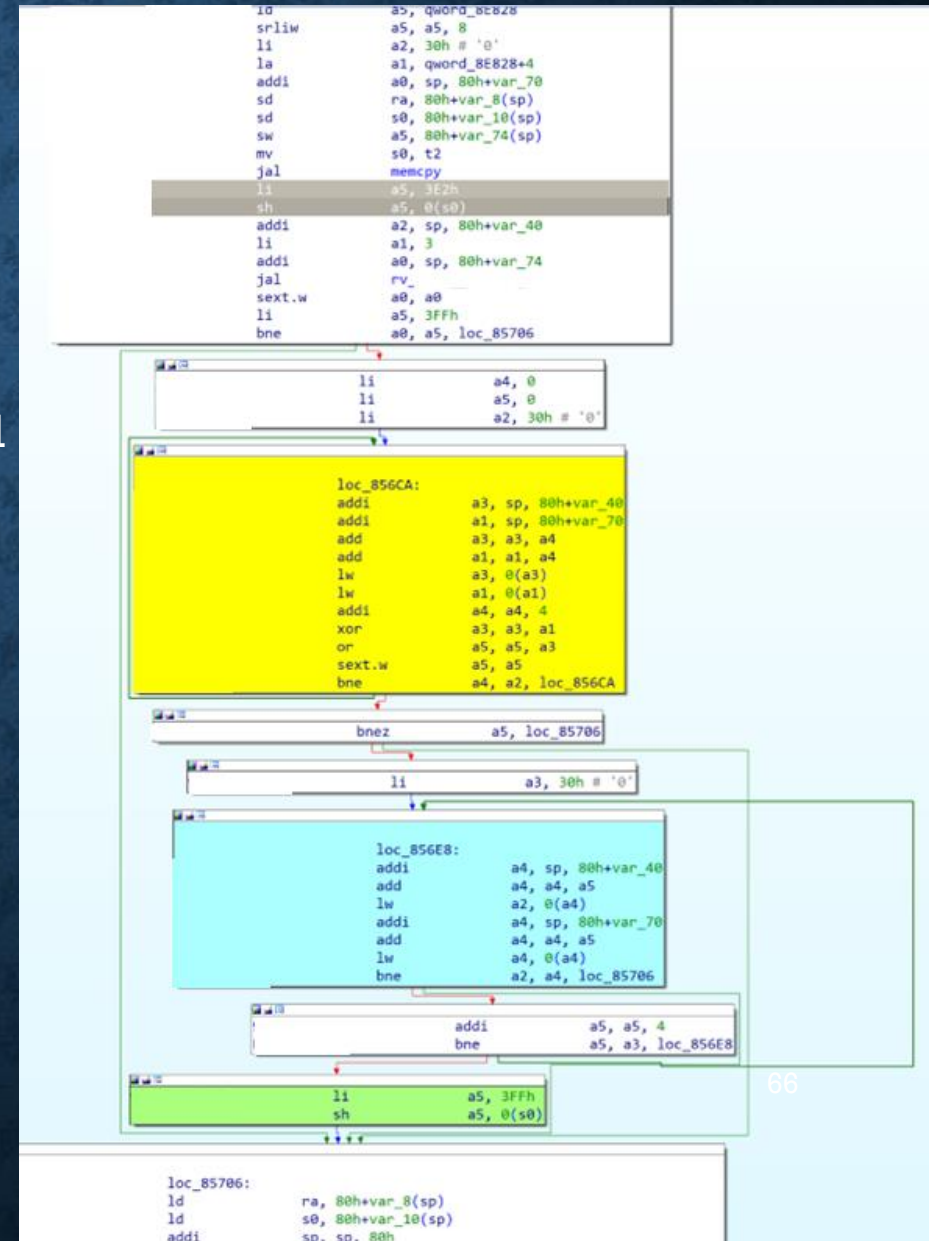


# EXAMPLES OF REAL-WORLD BUGS

## ❖ Problem with the compiler weakening FI protection

During code review, we observed redundant constant time loops to check equality for FI countermeasure robustness.

Additionally, we saw that the Hamming distance between pass and not pass states, was shortened



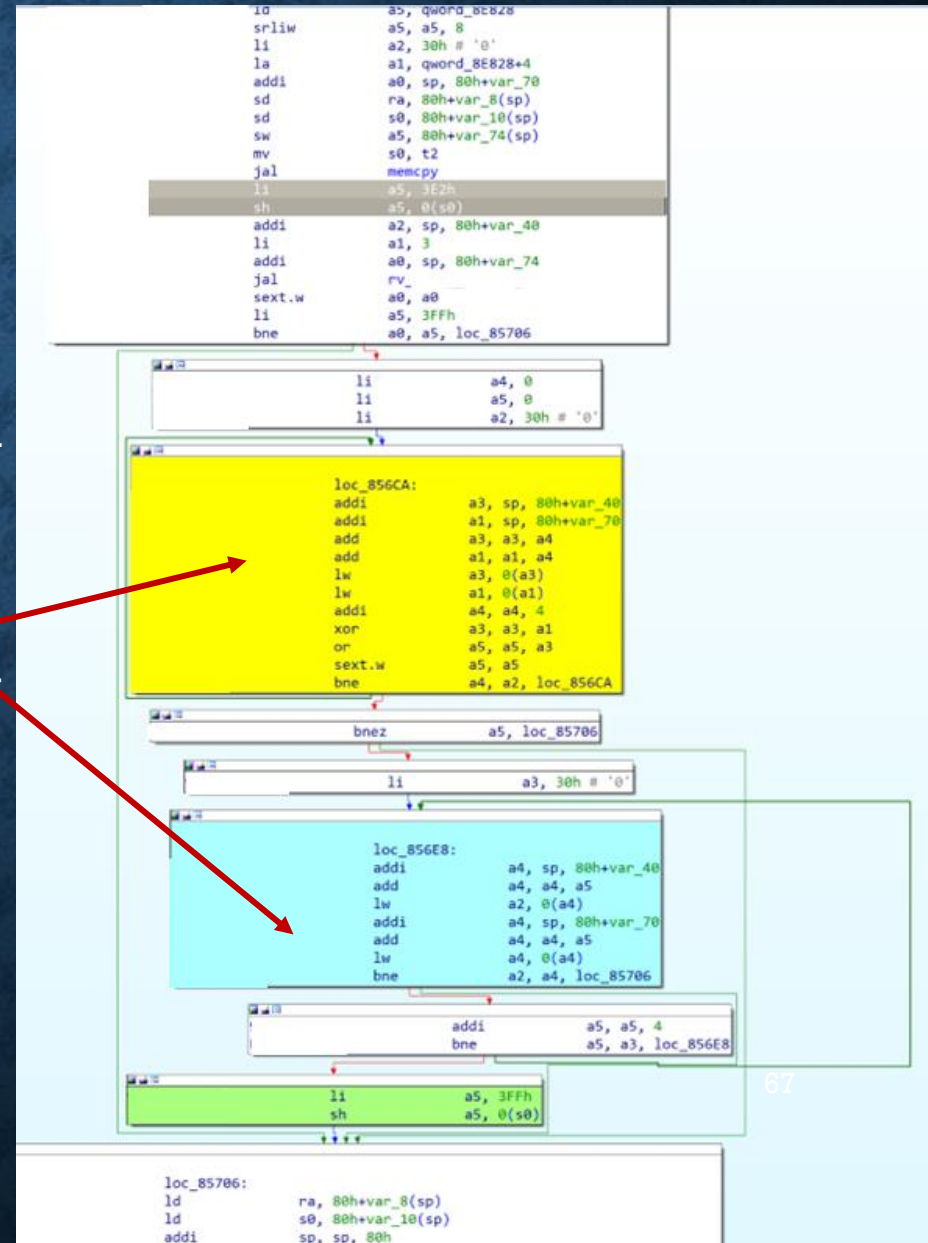
# EXAMPLES OF REAL-WORLD BUGS

## ❖ Problem with the compiler weakening FI protection

During code review, we observed redundant constant time loops to check equality for FI countermeasure robustness.

Additionally, we saw that the Hamming distance between pass and not pass states, was shortened

Constant-time  
Check of equality





# EXAMPLES OF REAL-WORLD BUGS

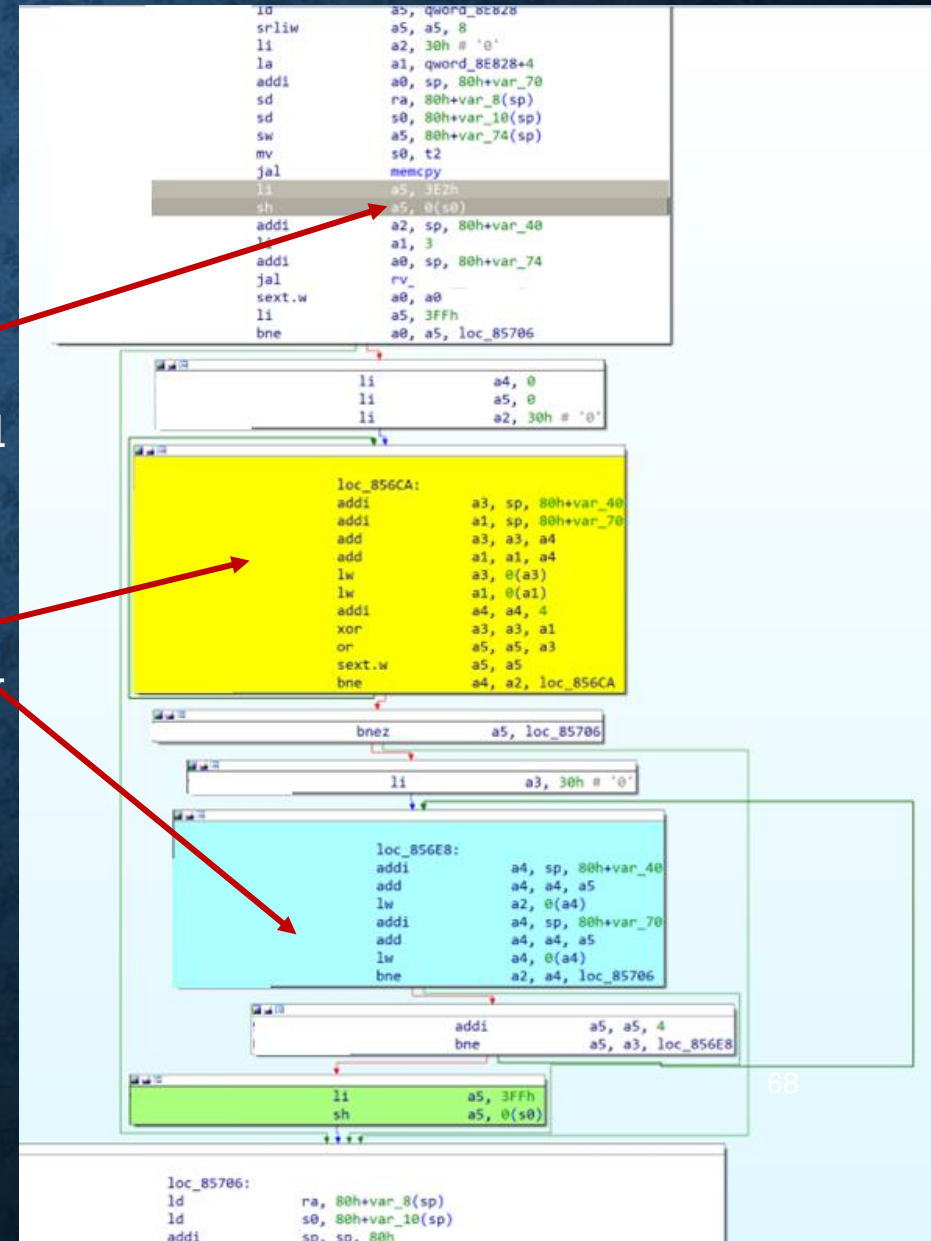
## ❖ Problem with the compiler weakening FI protection

During code review, we observed redundant constant time loops to check equality for FI countermeasure robustness.

Additionally, we saw that the Hamming distance between pass and not pass states, was shortened

Single point of weakness (a5)

Constant-time Check of equality





# EXAMPLES OF REAL-WORLD BUGS

## ❖ Problem with the compiler weakening FI protection

During code review, we observed redundant constant time loops to check equality for FI countermeasure robustness.

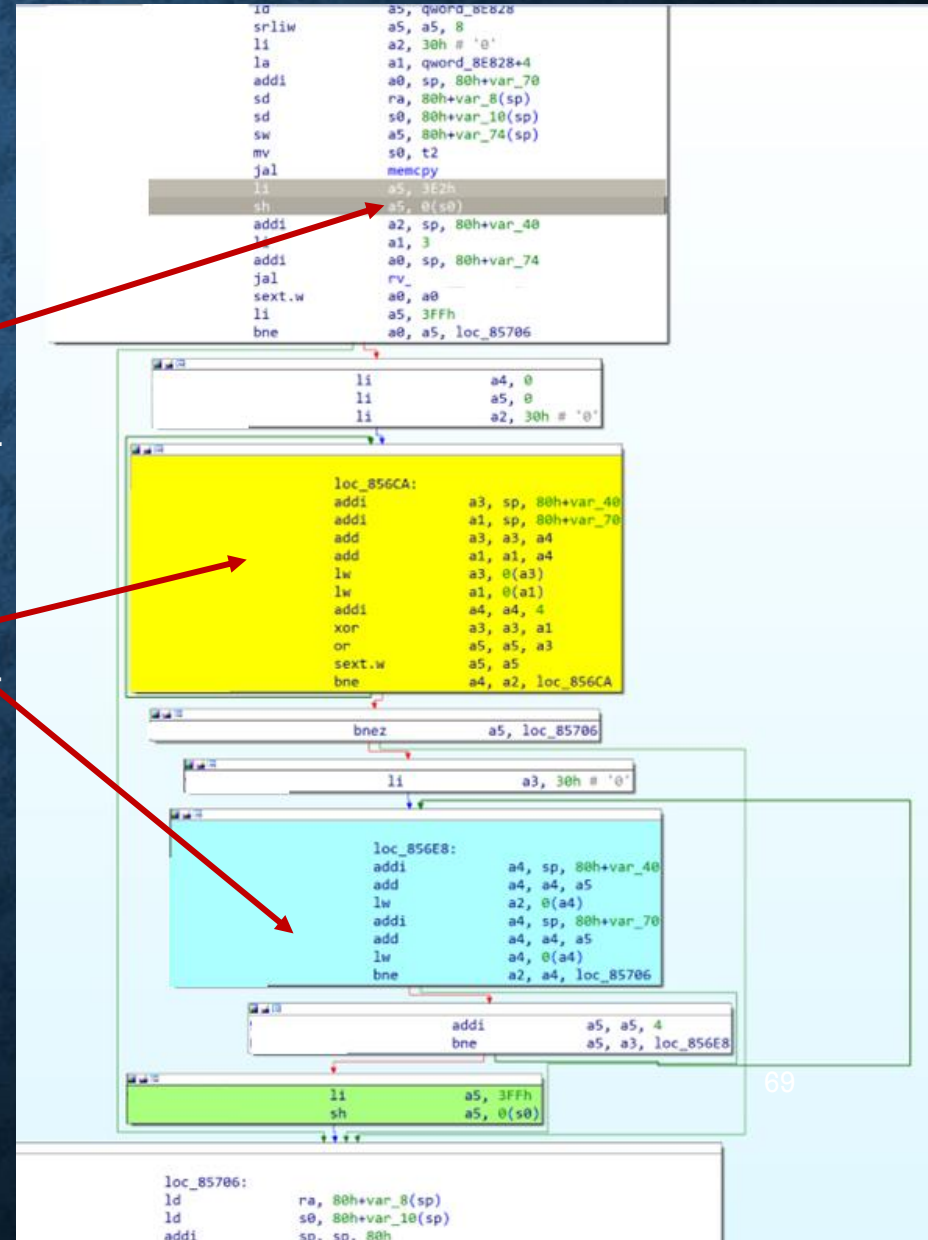
Additionally, we saw that the Hamming distance between pass and not pass states, was shortened

Single point of weakness (a5)

Constant-time Check of equality

Devs implemented critical function(s) in a way to be protected from FI (necessary for this product). Although, compiler optimized the code and inlined some of the functions weakening the protection.

- Compiler problem



# EXAMPLES OF REAL-WORLD BUGS

- ❖ Auto-init, AoRTE and design problem (mix of problems)



# EXAMPLES OF REAL-WORLD BUGS

## ❖ Auto-init, AoRTE and design problem (mix of problems)

```
893 procedure Get_Image_From_External_Media (...;
894     Err_Code : out ERROR_TYPE)
895 is
...
946 -- Sanitize for AORTE
947 if Header_Data.Total_Size <= 0 or else
948     Header_Data.Total_Size < Get_Size_Bytes (Size=> U32'Size) or else
949     Header_Data.Total_Size > Get_Size_Bytes (Size=> Buffer'Size) or else
950     Header_Data.Total_Size > U32 ((External_Media'Last) - Offset) or else
...
954
955 then
956
957     Err_Code := INVALID_IMAGE;
958
959     goto Exit;
960
961 end if;
962
963 Read_External_Image (Data => Buffer,
964     Size => Size_in_Bytes (Header_Data.Total_Size),
965     Offset=> Offset,
966     Media_Ctx=> Media_Ctx,
967     Err_Code => Err_Code);
```

SPARK can't prove correctness of metadata coming from untrusted source (E.g., external memory):

- However, developer can manually add verification and write contracts / conditions based on what was verified



# EXAMPLES OF REAL-WORLD BUGS

## ❖ Auto-init, AoRTE and design problem (mix of problems)

```
893 procedure Get_Image_From_External_Media (...;
894     Err_Code : out ERROR_TYPE)
895 is
...
946 -- Sanitize for AORTE
947 if Header_Data.Total_Size < U32'Size or else
948   Header_Data.Total_Size < Get_Size_Bytes (Size=> U32'Size) or else
949   Header_Data.Total_Size > Get_Size_Bytes (Size=> Buffer'Size) or else
950   Header_Data.Total_Size > U32 ((External_Media'Last) - Offset) or else
...
954
955 then
956
957   Err_Code := INVALID_IMAGE;
958
959   goto Exit;
960
961 end if;
962
963 Read_External_Image (Data => Buffer,
964   Size => Size_in_Bytes (Header_Data.Total_Size),
965   Offset=> Offset,
966   Media_Ctx=> Media_Ctx,
967   Err_Code => Err_Code);
```

SPARK can't prove correctness of metadata coming from untrusted source (E.g., external memory):

- However, developer can manually add verification and write contracts / conditions based on what was verified

# EXAMPLES OF REAL-WORLD BUGS

## ❖ Auto-init, AoRTE and design problem (mix of problems)

```
893 procedure Get_Image_From_External_Media (...;
894     Err_Code : out ERROR_TYPE)
895 is
...
946 -- Sanitize for AORTE
947 if Header_Data.Total_Size <= 0 or else
948   Header_Data.Total_Size < Get_Size_Bytes (Size=> U32'Size) or else
949   Header_Data.Total_Size > Get_Size_Bytes (Size=> Buffer'Size) or else
950   Header_Data.Total_Size > U32 ((External_Media'Last) - Offset) or else
...
954
955 then
956
957   Err_Code := INVALID_IMAGE;
958
959   goto Exit;
960
961 end if;
962
963 Read_External_Image (Data => Buffer,
964   Size => Size_in_Bytes (Header_Data.Total_Size),
965   Offset=> Offset,
966   Media_Ctx=> Media_Ctx,
967   Err_Code => Err_Code);
```

SPARK can't prove correctness of metadata coming from untrusted source (E.g., external memory):

- However, developer can manually add verification and write contracts / conditions based on what was verified
- Verified:
  - Maximum size of external media
  - Maximum size of local buffer
  - Minimum size
  - Some other important size related check (not shown)
  - Anything between X and Y passes the check



# EXAMPLES OF REAL-WORLD BUGS

## ❖ Auto-init, AoRTE and design problem (mix of problems)

```
893 procedure Get_Image_From_External_Media (...;
894     Err_Code : out ERROR_TYPE)
895 is
...
946 -- Sanitize for AORTE
947 if Header_Data.Total_Size <= 0 or else
948   Header_Data.Total_Size < Get_Size_Bytes (Size=> U32'Size) or else
949   Header_Data.Total_Size > Get_Size_Bytes (Size=> Buffer'Size) or else
950   Header_Data.Total_Size > U32 ((External_Media'Last) - Offset) or else
...
954
955 then
956
957   Err_Code := INVALID_IMAGE;
958
959   goto Exit;
960
961 end if;
962
963 Read_External_Image (Data => Buffer,
964   Size => Size_in_Bytes (Header_Data.Total_Size),
965   Offset=> Offset,
966   Media_Ctx=> Media_Ctx,
967   Err_Code => Err_Code);
```

SPARK can't prove correctness of metadata coming from untrusted source (E.g., external memory):

- However, developer can manually add verification and write contracts / conditions based on what was verified
- Verified:
  - Maximum size of external media
  - Maximum size of local buffer **Never goes outside media size**
  - Minimum size **Let's say 8**
  - Some other important size related check (not shown)
  - Anything between **X** and **Y** passes the check



# EXAMPLES OF REAL-WORLD BUGS

## ❖ Auto-init, AoRTE and design problem (mix of problems)

```
893 procedure Get_Image_From_External_Media(...;
894     Err_Code : out ERROR_TYPE)
895 is
...
946 -- Sanitize for AoRTE
947 if Header_Data.Total_Size <= 0 or else
948   Header_Data.Total_Size < Get_Size_Bytes(Size=> U32'Size) or else
949   Header_Data.Total_Size > Get_Size_Bytes(Size=> Buffer'Size) or else
950   Header_Data.Total_Size > U32'((External_Media'Last) - Offset) or else
...
954
955 then
956
957   Err_Code := INVALID_IMAGE;
958
959   goto Exit;
960
961 end if;
962
963 Read_External_Image(Data => Buffer,
964   Size => Size_in_Bytes(Header_Data.Total_Size),
965   Offset=> Offset,
966   Media_Ctx=> Media_Ctx,
967   Err_Code => Err_Code);
```

Image includes  
Header of 512  
bytes in size

SPARK can't prove correctness of metadata coming from untrusted source (E.g., external memory):

- However, developer can manually add verification and write contracts / conditions based on what was verified
- Verified:
  - Maximum size of external media
  - Maximum size of local buffer **Never goes outside media size**
  - Minimum size **Let's say 8**
  - Some other important size related check (not shown)
  - Anything between **X** and **Y** passes the check

# EXAMPLES OF REAL-WORLD BUGS

## ❖ Auto-init, AoRTE and design problem (mix of problems)

```
893 procedure Get_Image_From_External_Media(...;
894     Err_Code : out ERROR_TYPE)
895 is
...
946 -- Sanitize for AoRTE
947 if Header_Data.Total_Size <= 0 or else
948   Header_Data.Total_Size < Get_Size_Bytes(Size=> U32'Size) or else
949   Header_Data.Total_Size > Get_Size_Bytes(Size=> Buffer'Size) or else
950   Header_Data.Total_Size > U32'((External_Media'Last) - Offset) or else
...
954
955 then
956
957   Err_Code := INVALID_IMAGE;
958
959   goto Exit;
960
961 end if;
962
963 Read_External_Image(Data => Buffer,
964   Size => Size_in_Bytes(Header_Data.Total_Size),
965   Offset=> Offset,
966   Media_Ctx=> Media_Ctx,
967   Err_Code => Err_Code);
```

Image includes  
Header of 512  
bytes in size

SPARK  
enforces init\*.  
Zero uninit  
part of  
the buffer

SPARK can't prove correctness of metadata coming from untrusted source (E.g., external memory):

- However, developer can manually add verification and write contracts / conditions based on what was verified
- Verified:
  - Maximum size of external media
  - Maximum size of local buffer **Never goes outside media size**
  - Minimum size **Let's say 8**
  - Some other important size related check (not shown)
  - Anything between **X** and **Y** passes the check



# EXAMPLES OF REAL-WORLD BUGS

## ❖ Auto-init, AoRTE and design problem (mix of problems)

```
893 procedure Get_Image_From_External_Media(...)
894     Err_Code : out ERROR_TYPE)
895 is
...
946 -- Sanitize for AORTE
947 if Header_Data.Total_Size ... or else
948   Header_Data.Total_Size < Get_Size_Bytes(Size=> U32'Size) or else
949   Header_Data.Total_Size > Get_Size_Bytes(Size=> Buffer'Size) or else
950   Header_Data.Total_Size > U32 ((External_Media'Last) - Offset) or else
...
954 then
955
956   Err_Code := INVALID_IMAGE;
957
958   goto Exit;
959
960   end if;
961
962
963   Read_External_Image(Data => Buffer,
964   Size => Size_in_Bytes(Header_Data.Total_Size),
965   Offset=> Offset,
966   Media_Ctx=> Media_Ctx,
967   Err_Code => Err_Code);
```

Image includes  
Header of 512  
bytes in size

SPARK  
enforces init\*.  
Zero uninit  
part of  
the buffer

SPARK can't prove correctness of metadata coming from untrusted source (E.g., external memory):

- However, developer can manually add verification and write contracts / conditions based on what was verified
- Verified:
  - Maximum size of external media
  - Maximum size of local buffer **Never goes outside media size**
  - Minimum size **Let's say 8**
  - Some other important size related check (not shown)
  - Anything between **X** and **Y** passes the check

```
1601 procedure Verify_Signature(Id: Section_Id;
1602     Err_Code : out ERROR_TYPE)
1603 is
...
1628 begin
1629
1630   Err_Code := SUCCESS;
1631
1632   -- skip if already verified
1633   if Section_Map(Section_Id)=SIGNATURE_VERIFIED or else Id = 0
1634   then
1635
1636     then
1637
1638       goto Exit_Point;
1639
1640     end if;
...
1757   RSA_Verify(
...
1761     Err_Code => Err_Code);
1762
1763   -- Mark as verified
1764   if Err_Code = SUCCESS then
1765
1766     Section_Map(Section_Id):= SIGNATURE_VERIFIED;
1767
1768   end if;
1769
1770   <<Exit_Point>>
1771 end Verify_Signature;
```



# EXAMPLES OF REAL-WORLD BUGS

## ❖ Auto-init, AoRTE and design problem (mix of problems)

```

893 procedure Get_Image_From_External_Media(...)
894     Err_Code : out ERROR_TYPE)
895 is
...
946 -- Sanitize for AORTE
947 if Header_Data.Total_Size ... or else
948   Header_Data.Total_Size < Get_Size_Bytes(Size=> U32'Size) or else
949   Header_Data.Total_Size > Get_Size_Bytes(Size=> Buffer'Size) or else
950   Header_Data.Total_Size > U32 ((External_Media'Last) - Offset) or else
...
954 then
955     Err_Code := INVALID_IMAGE;
956
957     goto Exit;
958
959     goto Exit;
960
961 end if;
962
963 Read_External_Image(Data => Buffer,
964   Size => Size_in_Bytes(Header_Data.Total_Size),
965   Offset=> Offset,
966   Media_Ctx=> Media_Ctx,
967   Err_Code => Err_Code);

```

Image includes  
Header of 512  
bytes in size

SPARK  
enforces init\*.  
Zero uninit  
part of  
the buffer

SPARK can't prove correctness of metadata coming from untrusted source (E.g., external memory):

- However, developer can manually add verification and write contracts / conditions based on what was verified
- Verified:
  - Maximum size of external media
  - Maximum size of local buffer **Never goes outside media size**
  - Minimum size **Let's say 8**
  - Some other important size related check (not shown)
  - Anything between **X** and **Y** passes the check

```

1601 procedure Verify_Signature(Id: Section_Id;
1602     Err_Code : out ERROR_TYPE)
1603 is
...
1628 begin
1629
1630   Err_Code := SUCCESS;
1631
1632   -- skip if already verified
1633   if Section_Map(Section_Id) = SIGNATURE_VERIFIED or else Id = 0
1634   then
1635       then
1636
1637
1638   goto Exit_Point;
1639
1640   end if;
...
1757   RSA_Verify(
...
1761       Err_Code => Err_Code);
1762
1763   -- Mark as verified
1764   if Err_Code = SUCCESS then
1765       Section_Map(Section_Id) := SIGNATURE_VERIFIED;
1766
1767   end if;
1768
1769
1770   <<Exit_Point>>
1771 end Verify_Signature;

```

# EXAMPLES OF REAL-WORLD BUGS

## ❖ Auto-init, AoRTE and design problem (mix of problems)

```
893 procedure Get_Image_From_External_Media(...;
894     Err_Code : out ERROR_TYPE)
895 is
...
946 -- Sanitize for AORTE
947 if Header_Data.Total_Size < U32'Size or else
948   Header_Data.Total_Size < Get_Size_Bytes(Size=>U32'Size) or else
949   Header_Data.Total_Size > Get_Size_Bytes(Size=>Buffer'Size) or else
950   Header_Data.Total_Size > U32((External_Media'Last) - Offset) or else
...
954 then
955   Err_Code := INVALID_IMAGE;
956   goto Exit;
957 end if;
958
959 Read_External_Image(Data => Buffer,
960   Size => Size_in_Bytes(Header_Data.Total_Size),
961   Offset=> Offset,
962   Media_Ctx=> Media_Ctx,
963   Err_Code => Err_Code);
```

Image includes  
Header of 512  
bytes in size

SPARK  
enforces init\*.  
Zero uninit  
part of  
the buffer

SPARK can't prove correctness of metadata coming from untrusted source (E.g., external memory):

- However, developer can manually add verification and write contracts / conditions based on what was verified
- Verified:
  - Maximum size of external media
  - Maximum size of local buffer **Never goes outside media size**
  - Minimum size **Let's say 8**
  - Some other important size related check (not shown)
  - Anything between **X** and **Y** passes the check

```
1601 procedure Verify_Signature(Id: Section_Id;
1602     Err_Code : out ERROR_TYPE)
1603 is
...
1628 begin
1629
1630   Err_Code := SUCCESS;
1631
1632   -- skip if already verified
1633   if Section_Map(Section_Id) = SIGNATURE_VERIFIED or else Id = 0
1634   then
1635     goto Exit_Point;
1636   end if;
1637
1638   RSA_Verify(
1639     Err_Code => Err_Code);
1640
1641   -- Mark as verified
1642   if Err_Code = SUCCESS then
1643     Section_Map(Section_Id) := SIGNATURE_VERIFIED;
1644   end if;
1645
1646   <<Exit_Point>>
1647 end Verify_Signature;
```

# EXAMPLES OF REAL-WORLD BUGS

## ❖ Auto-init, AoRTE and design problem (mix of problems)

```
893 procedure Get_Image_From_External_Media (...)
894     Err_Code : out ERROR_TYPE)
895 is
...
946 -- Sanitize for AORTE
947 if Header_Data.Total_Size ... or else
948   Header_Data.Total_Size < Get_Size_Bytes (Size=> U32'Size) or else
949   Header_Data.Total_Size > Get_Size_Bytes (Size=> Buffer'Size) or else
950   Header_Data.Total_Size > U32 ((External_Media'Last) - Offset) or else
...
954 then
955   Err_Code := INVALID_IMAGE;
956   goto Exit;
957 end if;
958
959 Read_External_Image (Data => Buffer,
960                      Size => Size_in_Bytes (Header_Data.Total_Size),
961                      Offset=> Offset,
962                      Media_Ctx=> Media_Ctx,
963                      Err_Code => Err_Code);
```

Image includes  
Header of 512  
bytes in size

SPARK  
enforces init\*.  
Zero uninit  
part of  
the buffer

SPARK can't prove correctness of metadata coming from untrusted source (E.g., external memory):

- However, developer can manually add verification and write contracts / conditions based on what was verified
- Verified:
  - Maximum size of external media
  - Maximum size of local buffer **Never goes outside media size**
  - Minimum size **Let's say 8**
  - Some other important size related check (not shown)
  - Anything between **X** and **Y** passes the check

```
1601 procedure Verify_Signature (Id : Section_Id;
1602                               Err_Code : out ERROR_TYPE)
1603 is
...
1628 begin
1629
1630   Err_Code := SUCCESS;
1631
1632   -- skip if already verified
1633   if Section_Map (Section_Id) = SIGNATURE_VERIFIED or else Id = 0
1634   then
1635     goto Exit_Point;
1636   end if;
1637
1638   RSA_Verify (
1639     Err_Code => Err_Code);
1640
1641   -- Mark as verified
1642   if Err_Code = SUCCESS then
1643     Section_Map (Section_Id) := SIGNATURE_VERIFIED;
1644   end if;
1645
1646   <<Exit_Point>>
1647 end Verify_Signature;
```

Skipped



# EXAMPLES OF REAL-WORLD BUGS

- ❖ Lack of memory isolation between U and S mode (RISC-V)

# EXAMPLES OF REAL-WORLD BUGS

- ❖ Lack of memory isolation between U and S mode (RISC-V)

```
64  Cfg.Access_Type := (S_Read => 16#1#,  
65                      S_Write => 16#0#,  
66                      S_Exec => 16#1#,  
67                      U_Read => 16#1#,  
68                      U_Write => 16#0#,  
69                      U_Exec => 16#1#);  
70  Cfg.Start_Offset := REGION_START;  
71  Cfg.Region_Rng := REGION_SIZE;  
...
```

SPARK doesn't know the context:

- Prover didn't catch that
- It's a logical error

# EXAMPLES OF REAL-WORLD BUGS

- ❖ Lack of memory isolation between U and S mode (RISC-V)

```
64  Cfg.Access_Type := {S_Read => 16#1#,  
65                      S_Write => 16#0#,  
66                      S_Exec => 16#1#,  
67                      U_Read => 16#1#,  
68                      U_Write => 16#0#,  
69                      U_Exec => 16#1#};  
70  Cfg.Start_Offset := REGION_START;  
71  Cfg.Region_Rng := REGION_SIZE;  
...
```

SPARK doesn't know the context:

- Prover didn't catch that
- It's a logical error
- it's possible to “build” such knowledge via Ghost code but it's difficult and very expensive (model entire HW behavior, registers meaning, etc.)

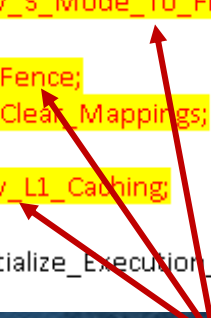


# EXAMPLES OF REAL-WORLD BUGS

## ❖ Lack of memory isolation between U and S mode (RISC-V)

```
64  Cfg.Access_Type := (S_Read => 16#1#,
65                      S_Write => 16#0#,
66                      S_Exec => 16#1#,
67                      U_Read => 16#1#,
68                      U_Write => 16#0#,
69                      U_Exec => 16#1#);
70  Cfg.Start_Offset := REGION_START;
71  Cfg.Region_Rng := REGION_SIZE;
...
```

```
xx5  procedure Initialize_Execution_Environment(...)
xx6  is
xx7    procedure Cpu_Reg_Wr64 is new Cpu.Wr64 (Generic_Reg => CPU_...);
xx8  begin
...
yx7    Allow_S_Mode_To_Flush_D_Cache;
...
yx8    CPU.Fence;
yx9    CPU.Clear_Mappings;
...
zx1    Allow_L1_Caching;
...
yx4  end Initialize_Execution_Environment;
```



SPARK doesn't know the context:

- Prover didn't catch that
- It's a logical error
- it's possible to "build" such knowledge via Ghost code but it's difficult and very expensive (model entire HW behavior, registers meaning, etc.)

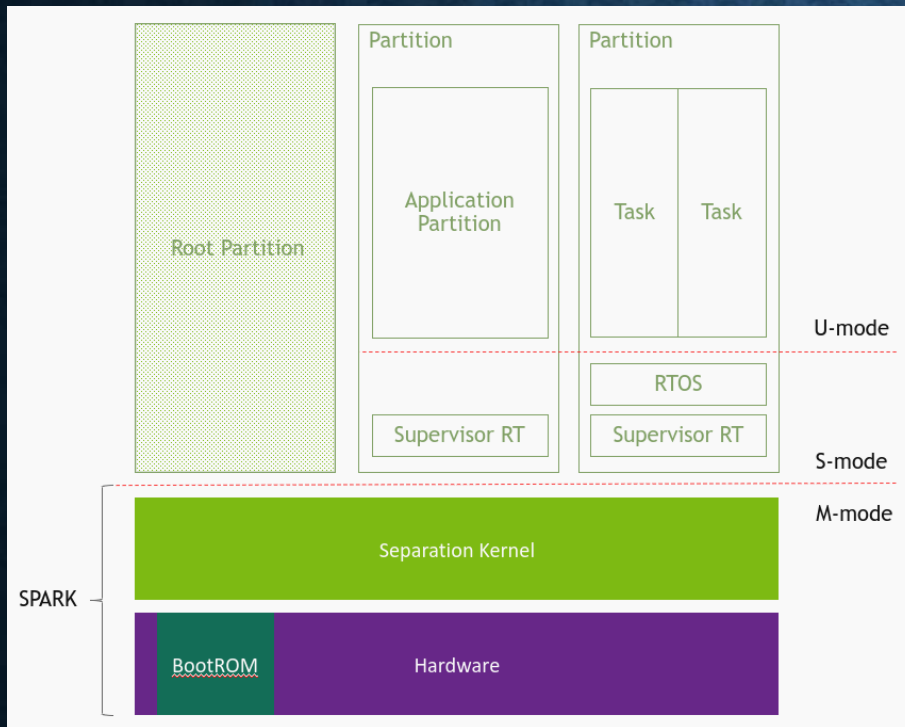
D-cache is **NOT** invalidated, and lower-privileged mode has an ability to self-manage Caches – it leaves the door open for **side-channel attacks**

# EXAMPLES OF REAL-WORLD BUGS

- ❖ Shared memory problem in isolation partitions (RISC-V)

# EXAMPLES OF REAL-WORLD BUGS

- ❖ Shared memory problem in isolation partitions (RISC-V)

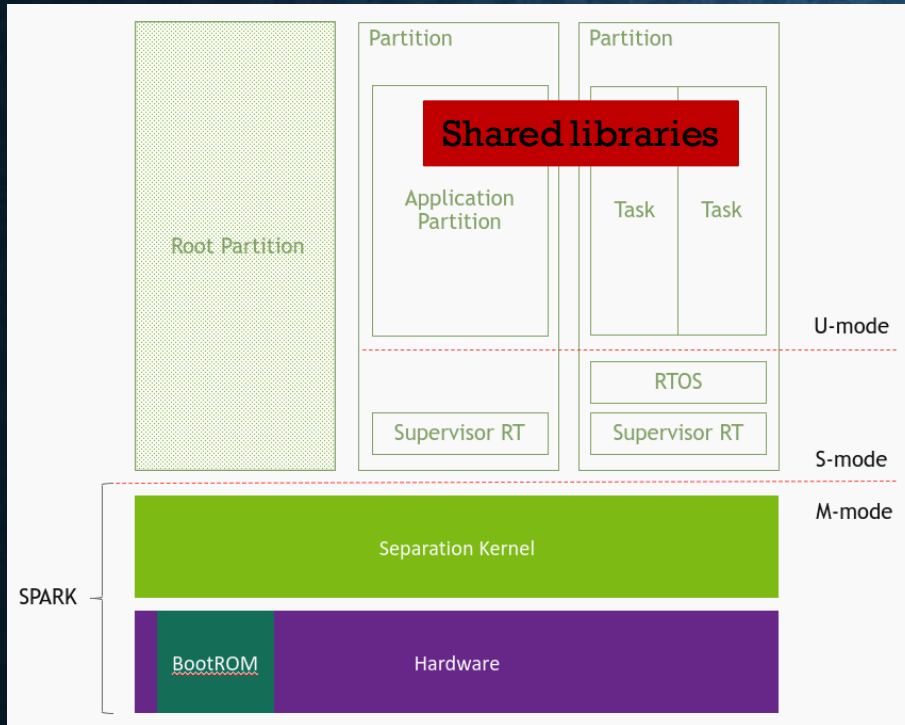


<https://youtu.be/17i1kfHvWNI>



# EXAMPLES OF REAL-WORLD BUGS

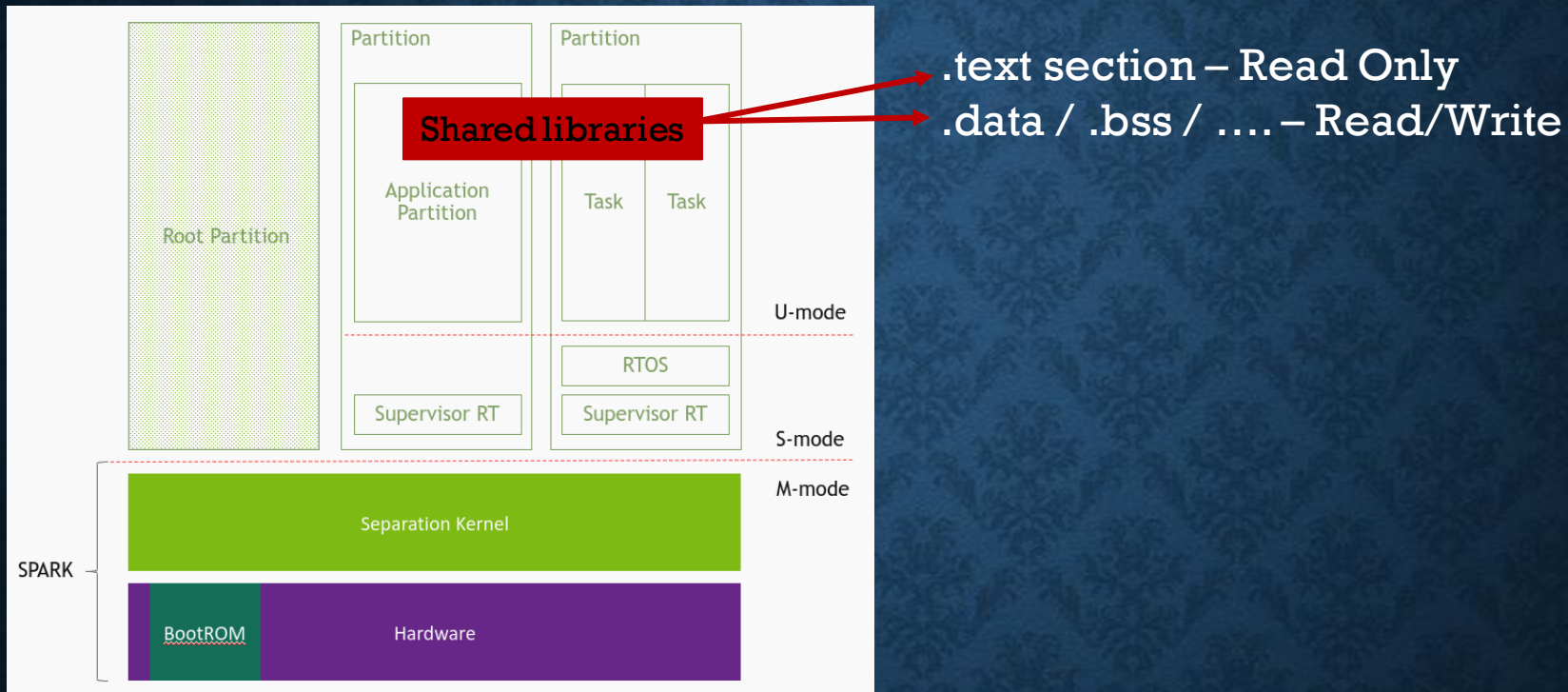
- ❖ Shared memory problem in isolation partitions (RISC-V)



<https://youtu.be/17ilkfHvWNI>

# EXAMPLES OF REAL-WORLD BUGS

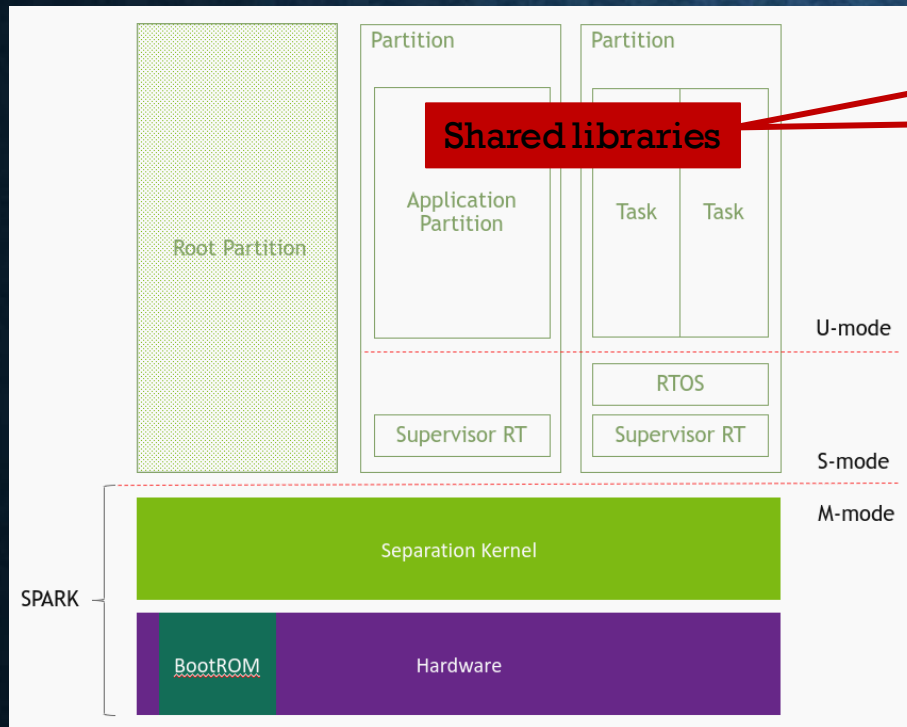
## ❖ Shared memory problem in isolation partitions (RISC-V)



<https://youtu.be/17ilkfHvWNI>

# EXAMPLES OF REAL-WORLD BUGS

## ❖ Shared memory problem in isolation partitions (RISC-V)



.text section – Read Only  
.data / .bss / .... – Read/Write

Might include:

- Global state
- Initialization data
- Control flow data
- Pointers
- More...

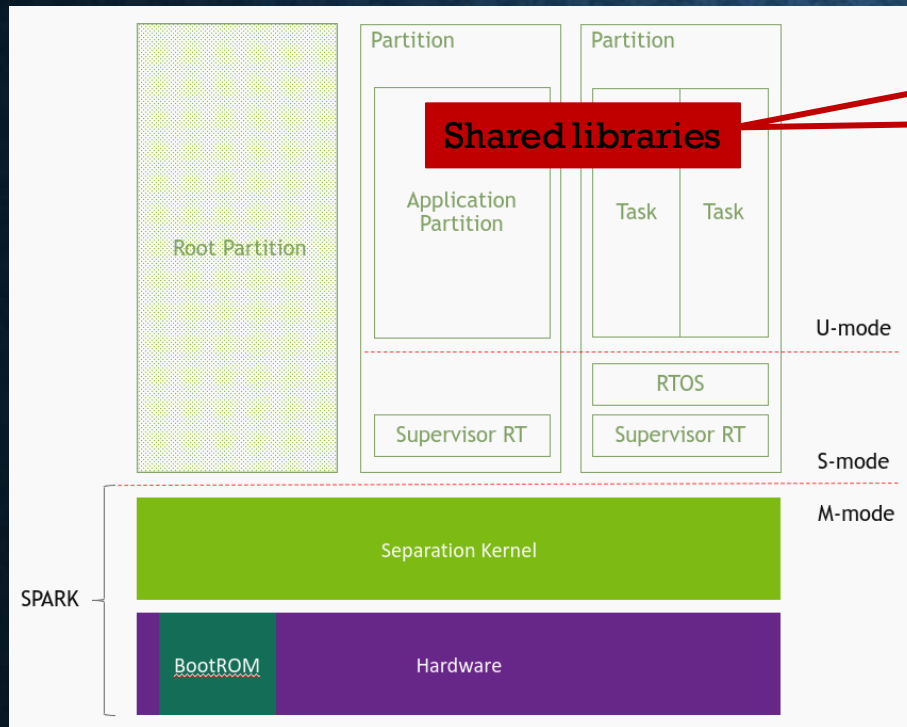
Especially in  
non-SPARK code

<https://youtu.be/17ilkfHvWNI>



# EXAMPLES OF REAL-WORLD BUGS

## ❖ Shared memory problem in isolation partitions (RISC-V)



.text section – Read Only  
.data / .bss / .... – Read/Write

Might include:

- Global state
- Initialization data
- Control flow data
- Pointers
- More...

Especially in  
non-SPARK code

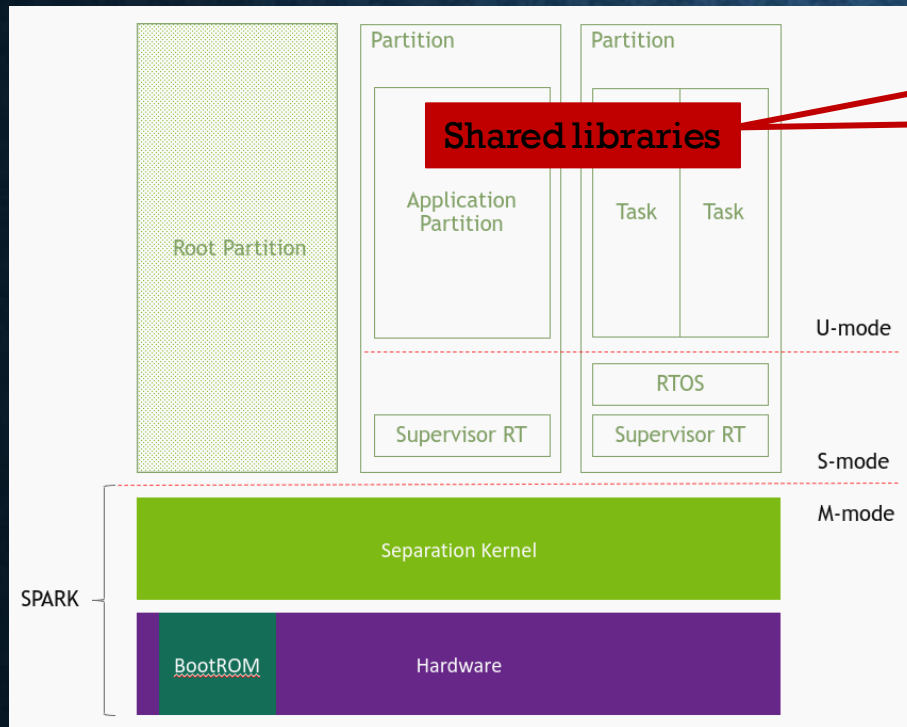
SPARK doesn't know the context:

- Prover won't catch that
- It's a context-related logical error

<https://youtu.be/17i1kfHvWNI>

# EXAMPLES OF REAL-WORLD BUGS

## ❖ Shared memory problem in isolation partitions (RISC-V)



.text section – Read Only  
.data / .bss / .... – Read/Write

Might include:

- Global state
- Initialization data
- Control flow data
- Pointers
- More...

Especially in  
non-SPARK code

SPARK doesn't know the context:

- Prover won't catch that
- It's a context-related logical error

<https://youtu.be/17ilkfHvWNI>



# EXAMPLES OF REAL-WORLD BUGS



Insert  
exploit  
video here  
;-)



# ACKNOWLEDGMENTS

❖ We would like to thank:

❖ NVIDIA:

❖ Offensive Security Research team:

Jared Candelaria, Max Bazalii, Nikola Livic

❖ GPU System Software:

James Xu, Marko Mitic, Mateusz Kulikowski, RISC-V SW team,  
Varun Sampath, Shreyas Shyamsunder, Steven Bellock

❖ Product Security:

Shawn Richardson, PSIRT team, ThreatOps team(s)

❖ Everyone :)

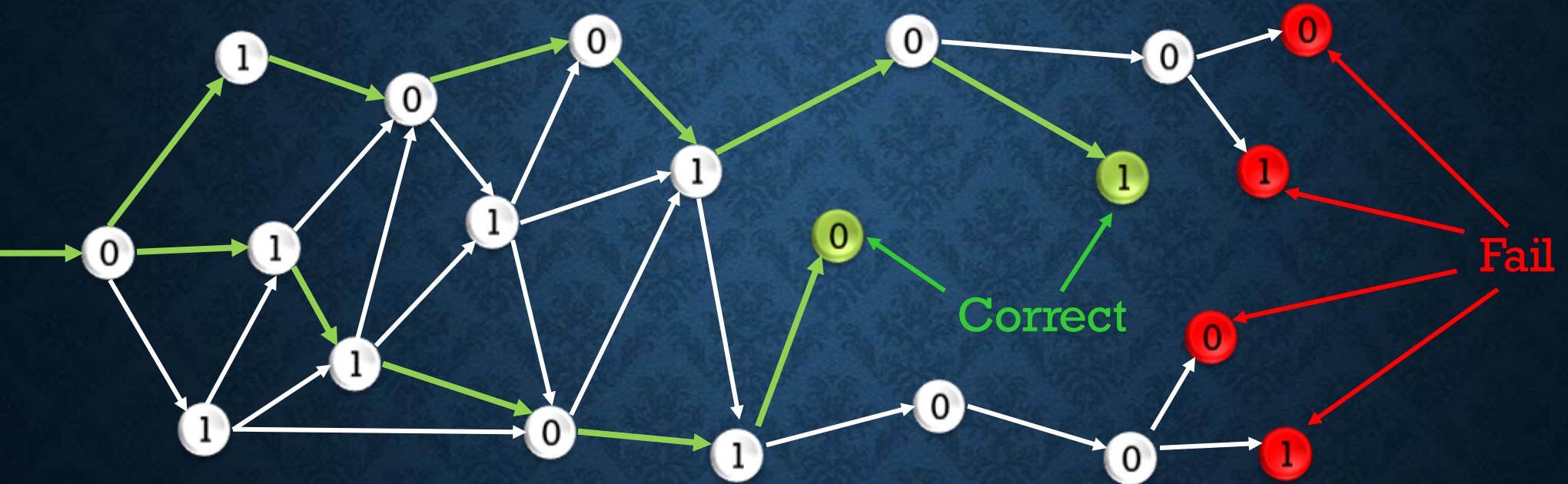
❖ AdaCore

# SUMMARY

- ❖ The use of Type Safety languages and Formal Verification minimizes the attack surfaces not only for memory corruption issues, but it is not a silver bullet
- ❖ Formally verified software has much higher quality thanks to SPARK enforcements
- ❖ SPARK may prove that dynamic checks cannot fail (AoRTE)
  - ❖ Depends on the levels of assurance (Silver+)
- ❖ Enables more efficient offensive security efforts
- ❖ Most of the bugs which we saw requires deep knowledge and understanding not only the software but also hardware (more “deep” bugs, architecture problems, design bugs, etc):
  - ❖ OSR review of projects in memory unsafe languages – ~40-50 bugs in 4 weeks
  - ❖ OSR review of projects in SPARK – ~5-10 bugs in 6 weeks – better “quality” of bugs :)<sup>94</sup>



# Q&A



## Private contact:

<http://pi3.com.pl>

pi3@pi3.com.pl

Twitter: [@Adam\\_pi3](#)

# Adam 'pi3' Zabrocki



## Private contact:

[alex.tereshkin@gmail.com](mailto:alex.tereshkin@gmail.com)

Twitter: [@AlexTereshkin](https://twitter.com/AlexTereshkin)

# Alex Tereshkin